7
DIGITAL & SOLID STATE APPENDICES

Reliability-1

Concept and definitions

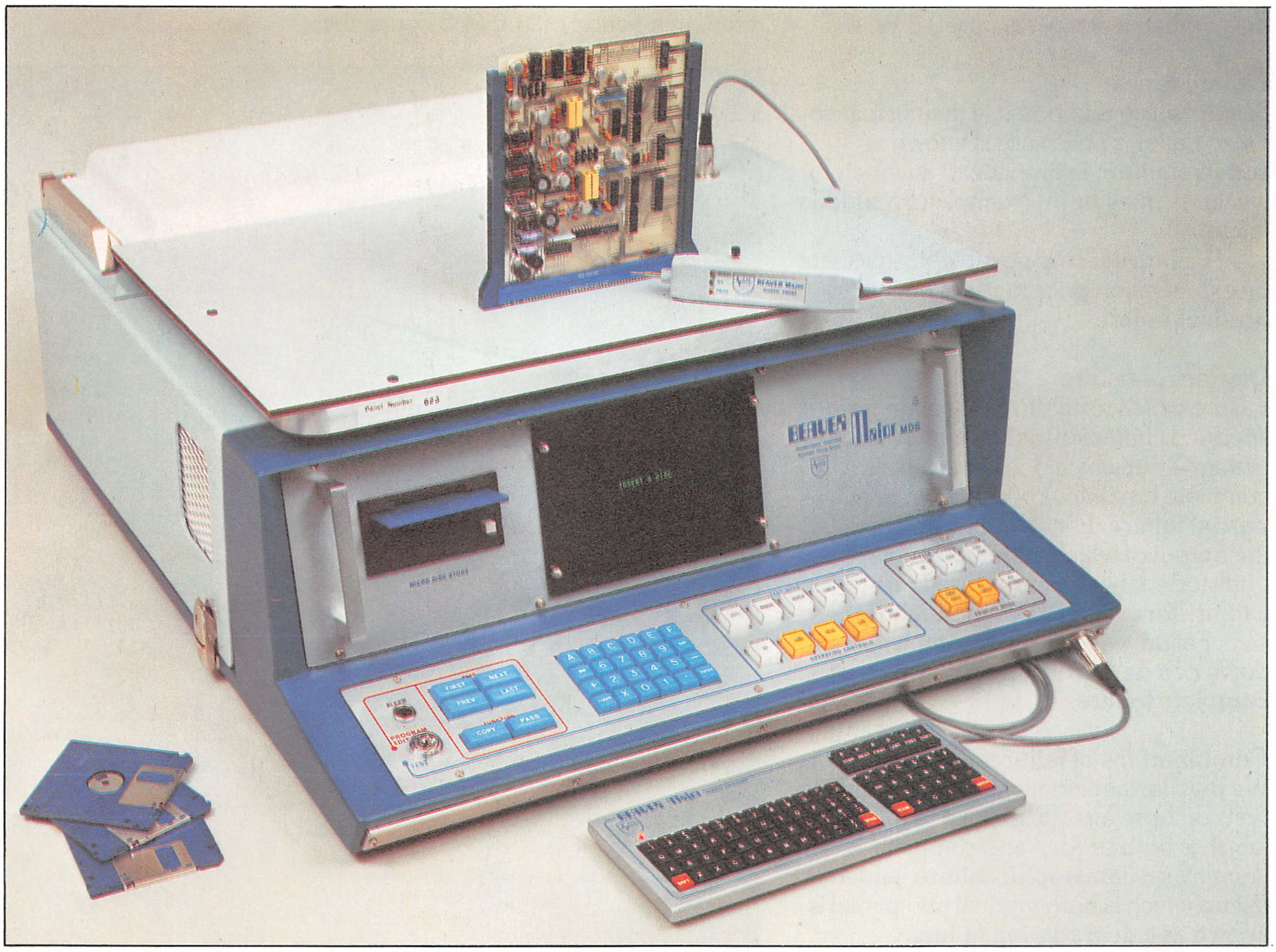
The rapidly increasing complexity of modern electronics systems – both digital and analogue – has demanded that they are as accurate and as reliable as possible.

Unfortunately, as complexity increases, reliability generally becomes more difficult to achieve, because the reliability of a whole system relies on the reliability of each part – a chain is only as strong as its weakest link. Nevertheless, the reliability of electronic systems has improved considerably since the days when systems frequently broke down and maintenance costs were about 10 times the purchase price.

We may define the reliability of any electronic device or system as its ability to perform its required function, under known conditions for a certain period of time. The definition requires that we must define, for any electronic device or system:
1) its function, or functions;
2) the conditions under which it operates;
3) the length of time considered.

Any device or system may, at some

Below: BEAVER Major MDS automatic testing system for electronic circuits. (Photo: ATE Systems Ltd).



time, break down or **fail**. We may define **failure** as: the termination of a device or system's ability to perform its required function. The process which causes failure is known as the **failure mechanism**.

Failure itself may be further defined in a number of ways, depending on the cause, timing or degree of failure.

Causes of failure

Failure which is attributed to the application of stresses beyond the stated capabilities of the device or system, for example, too high a voltage applied across a capacitor causing the dielectric to breakdown, is known as **misuse failure**.

Failure which is attributed to weakness inherent within the device or system when subjected to a stress or stresses within its stated capability is known as **inherent weakness failure**. Such failure could be caused, for example, if a resistor, rated to dissipate a power of 1 W, breaks down when it dissipates only 0.5 W.

Failure times

Failure which is impossible to anticipate by examination prior to use is known as **sudden failure**; failure due to an inherent weakness may be an example of a sudden failure.

Failure which *is* possible to anticipate by examination prior to use is known as **gradual failure**.

Degrees of failure

Failure which results in a deviation from specified characteristic limits of a device or system – but which does not cause a complete breakdown – is known as a **partial failure**. For example: an amplifier, the output of which, although originally undistorted, becomes distorted due to, say, a transistor with an inherent weakness.

Failure which causes complete breakdown of a device or system is known as **complete failure**.

Combinations of failures

We may further define failures as *combinations* of those already defined: a failure which is both sudden and complete is known as a **catastophic failure**; and a failure which is both gradual and partial is known as a **degradation failure**.

Quantitative measurements of failure

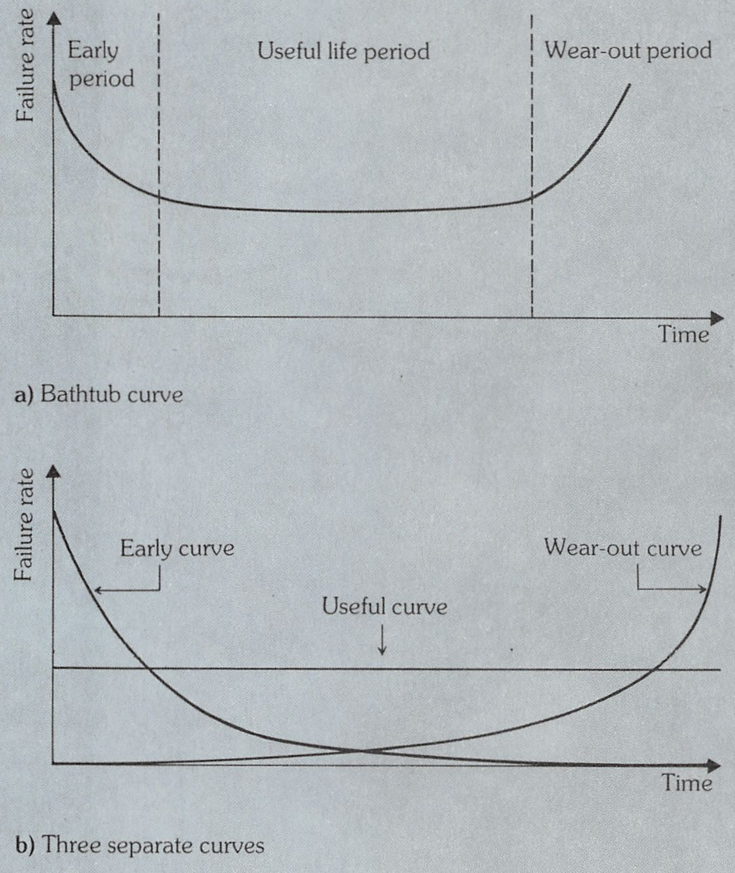
Once the reliability of a device or system, and failure, have been defined, the most important aspect is knowledge of how often failure may be expected. This is measured as **mean time between failure** (MTBF), where:

$$\text{MTBF} = \frac{T}{N}$$

where T is the total operating time in hours (excluding repair time) and N is the number of times operation is interrupted. The MTBF may be derived experimentally, by operating the device or system under the specified conditions, and then calculating the mean value of the times between each failure. Obviously, a number of failures must be observed in order that an accurate figure of mean time is obtained. For example, if the device or system fails 10 times in a period of 100,000 hours, the

1. (a) Failure rate *vs* time curve – known as the bathtub curve which; **(b)** is made of from the three separate curves – early, useful and wear out.

1



MTBF is 10,000 hours.

Availability

The period of time a device or system is operational is known as its **availability** – this is dependent on the number of failures occurring during the device or system's life, and also on the time taken to repair each fault.

Maintainability

The time taken to repair a system which has failed is known as the system's **maintainability**, expressed as the **mean time to repair** (MTTR) in hours. MTTR may be reduced by considering the factors involved in the repair of a failed system, for example:

1) the time taken for a service engineer to reach the system;
2) the time taken to isolate the fault;
3) the time taken to repair or replace the faulty component(s);
4) the time taken to check and verify that the system is working correctly.

By ensuring that: sufficient service engineers are available; the engineers are well trained; the system is well documented; maintenance check points are included in the system; the engineers have adequate test equipment; spare parts are available, it is possible for a manufacturer to reduce the system's mean time to repair and thus increase its maintainability.

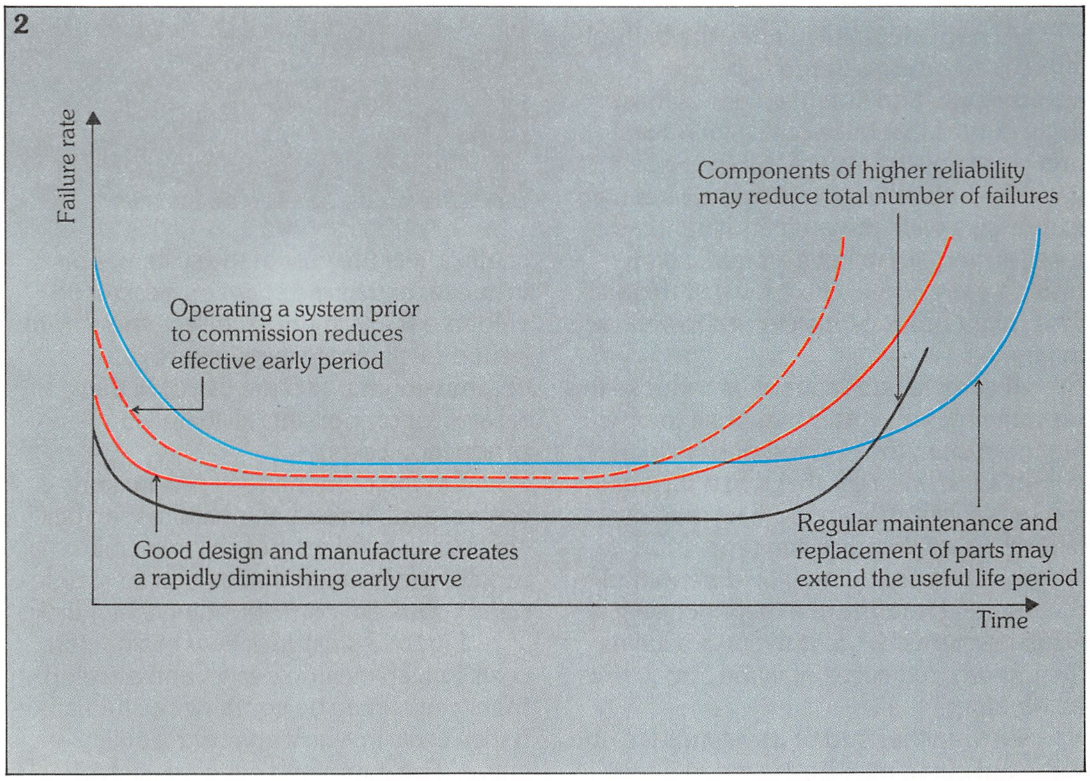
Failure and time

The curve in *figure 1a* shows **failure rate** (the number of failures occuring in unit time) plotted against time and is typical of most electronic devices and systems. Because of its shape, the curve is sometimes called a **bathtub curve**. Three distinct areas can be identified on the graph: the early period; the useful life period; and the wear-out period.

The bathtub curve illustrates that for a short period when electronic systems are first commissioned, the failure rate is high. Such **early failures** may be due to substandard components or substandard workmanship in manufacture.

During the **useful life period**, the failure rate is more or less constant, with failures occurring at more or less predictable intervals. Failures in this period are due to random faults.

It is possible to predict a system's reliability during the useful life period:



2. Controlling the shape of the bathtub curve.

1) from a knowledge of the system's component reliability;
2) tests made on a similar system.

The failure rate increases during the **wear-out period** due to general deterioration of the system through component age.

We can see from *figure 1b* that the overall bathtub curve may be considered as a combination of three separate curves corresponding to the failures in each period: the **early curve** – diminishing with time; the **useful curve** – constant with time; the **wear-out curve** – increasing with time.

At any one point in time, the bathtub curve may be considered to be the approximate addition of values of these three curves. Because of this, manufacturer's are able to control the shape of a system's bathtub curve to some extent. For example, a well designed and manufactured system, may have a rapidly diminishing early curve with a low failure rate. This might produce the curve shown in red in *figure 2*.

If early failures cannot be reduced this way, the manufacturer may wish to operate the system for a period prior to commission, in order that when the customer is supplied with the system, early failures have already been repaired. The length of this period may be determined by the manufacturer from experience with similar systems. A bathtub curve such as the broken, red curve of *figure 2* may thus be obtained.

At the other end of a system's life, the wear-out curve may be extended if regular maintenance and replacement of the system's component parts is undertaken throughout the useful life period. In such a case, the overall bathtub curve may look like the blue curve of *figure 2*, which illustrates how the useful life of the system has been extended.

Finally, the total number of failures may be reduced, for example, by using components of higher reliability. The black curve of *figure 2* shows how the bathtub curve may thus be affected.

Reliability and cost

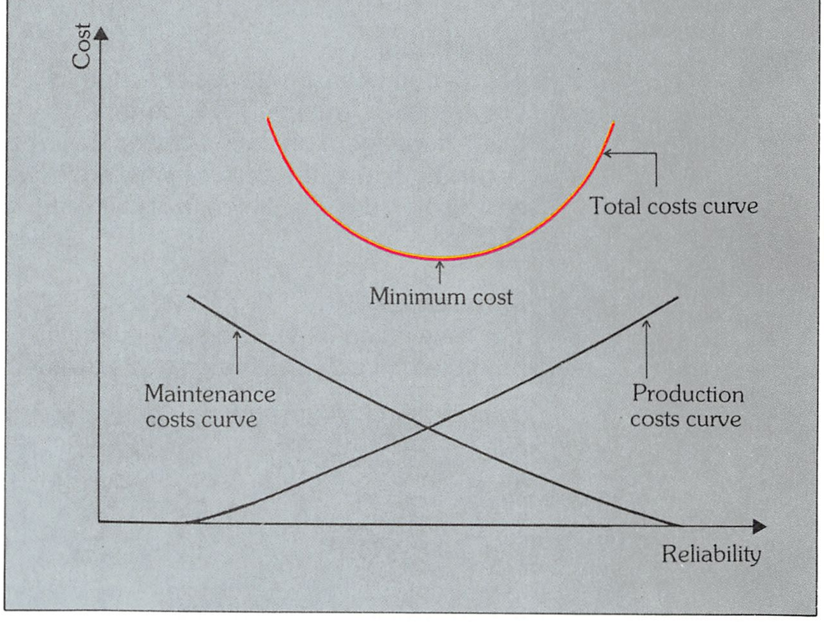
As we have seen there are a number of ways in which manufacturers can improve system reliability. However, this can significantly affect the cost of the system. For instance: extra time spent on design, development and manufacture; a trial run-in period to eliminate early failures after commissioning; and the use of higher reliability components all increase production costs.

On the other hand, if production costs are minimised, the system may be cheaper to produce but will probably require more maintenance and so maintenance costs are correspondingly higher.

Figure 3 illustrates how production costs (which include design and development costs) may be combined with maintenance costs in a cost against reliability curve. The combined effect of each of these two curves is the total cost curve, shown in red. From this total cost curve we can see that there is a minimum point where the system cost is at its lowest. The manufacturer may wish to produce the system at this cost, knowing the production costs and the maintenance costs will be at their lowest for reasonable reliability.

On the other hand, the manufacturer may opt for increased reliability and hence reduced maintenance costs – however, as production costs correspondingly increase, the total system cost is also greater. This, however, may be beneficial in the long term, as greater reliability increases customer confidence, hence more systems may be sold.

3



3. Cost *vs* reliability curve producing a minimum cost.

Calculating reliability

The reliability of a complete system may be simply calculated in a two-stage process.
1) Add together the failure rates of each component in the system, to find the system failure rate, so that:

$$\lambda_s = \lambda_1 + \lambda_2 + \lambda_3 + \dots \lambda_n$$

2) Calculate the MTBF as:

$$\text{MTBF} = \frac{1}{\lambda_s}$$

The failure rate of each component in the system must be calculated with regard to any operating stresses and environmental conditions which may affect it.

Generally, component failure rates, and **weighting factors** for operating stresses and environmental conditions, are supplied by the component manufacturer. The system manufacturer simply compiles the data to find the system failure rate.

As an example, suppose that a transistor radio is to be manufactured, which will contain: 50 transistors operating at 0.5 of their maximum rating; 50 carbon film resistors operating at 0.1 of their maximum rating; 50 carbon composition resistors operating at maximum rating; 50 electrolytic capacitors operating at maximum rating; 6 coils operating at 0.1 maximum rating; and 500 connections. The radio is for home use in the U.K.

Table 1 lists some possible failure rates of the components used in the radio, used at 0.1 of their maximum rating in a home-type environment. *Table 2* lists possible weighting factors due to a range of operating stresses. From these tables we can calculate the various failure rates of each group of components.

The failure rate of the 50 transistors at 0.5 of their maximum rating is:

$$50 \times 1 \times 10^{-7} \times 1.5 = 7.5 \times 10^{-6}\,\text{h}^{-1}$$

The failure rate of the 50 carbon film resistors at 0.1 of their maximum rating is:

$$50 \times 1 \times 10^{-6} = 50 \times 10^{-6}\,\text{h}^{-1}$$

The carbon composition resistors' failure rate is:

$$50 \times 5 \times 10^{-8} \times 2 = 5 \times 10^{-6}\,\text{h}^{-1}$$

The electrolytic capacitors' failure rate is:

$$50 \times 2 \times 10^{-6} \times 6 = 600 \times 10^{-6}\,\text{h}^{-1}$$

The failure rate of the coils is:

$$6 \times 5 \times 10^{-7} = 3 \times 10^{-6}\,\text{h}^{-1}$$

Finally, the failure rate of the 500 connections is given by:

$$500 \times 1 \times 10^{-8} = 5 \times 10^{-6}\,\text{h}^{-1}$$

Table 1
Component failure rates at 0.1 rating

Components	Failure rate (h^{-1})
Capacitors, electrolytic	2×10^{-6}
Coils	5×10^{-7}
Connections	1×10^{-8}
Resistors, carbon composition	5×10^{-8}
Resistors, carbon film	1×10^{-6}
Transistors	1×10^{-7}

Table 2
Weighting factors due to a range of operating stresses

Component	Rating	Weighting factor
Capacitors	0.1 maximum	1
	0.5 maximum	3
	maximum	6
Resistors	0.1 maximum	1
	0.5 maximum	1.5
	maximum	2
Semiconductors	0.1 maximum	1
	0.5 maximum	1.5
	maximum	2

Total system failure rate is therefore:

$$\lambda_s = (7.5 + 50 + 5 + 600 + 3 + 5) \times 10^{-6}\,h^{-1}$$
$$= 670.5 \times 10^{-6}\,h^{-1}$$

The mean time between failures of our example transistor radio is given by:

$$\text{MTBF} = \frac{1}{\lambda_s}$$
$$= \frac{1}{670.5 \times 10^{-6}}$$
$$= 1491\ h$$

Failure rate of a component or system is normally stated as a percentage per 1000 hours. For example, the transistor radio with a failure rate of:

$$\lambda_s = 670.5 \times 10^{-6}\,h^{-1}$$

would be stated as:

$$\lambda_s = 67.05\ \%\ 1000\,h^{-1}$$

using these units. The first number has simply been multiplied by 100,000 to obtain the second.

Measuring reliability

It is possible to determine a system's reliability by testing it under controlled conditions. Generally, a single system under test will not give results representative of other identical systems and it is usual, therefore, to test a sample number of systems. The greater the number of samples, the more representative the results are likely to be.

If the controlled conditions are such that they simulate normal environments, e.g. home use, the system under test may only fail on rare occasions (the transistor radio example MTBF was 1491 hours, remember – and this is by no means a high figure). A transistor radio is used for only a few hours a week, say 6 h week^{-1}, so the expected life before failure is:

$$\frac{1491}{6} = 248.5\ \text{weeks}$$

or, over 4.5 years. A system which is in constant use, on the other hand, must have a much greater MTBF. This means that a sample test to determine a system's reliability, with these conditions will obviously take quite a long time to conduct.

We already know, however, that failure rates of components are affected by the environmental conditions under which the components operate. This means that a manufacturer may alter the system's environmental conditions, in turn, altering the failure rate. A very harsh operating environment of, say, high humidity or high temperature, will increase the system failure rate.

Knowing the weighting factor which the harsh environment imposes on each component within the system, the manufacturer may then recalculate the **accelerated failure rate** obtained by test, to give the expected failure rate under normal conditions.

Many types of **test chambers** or machines are commonly used to test systems in this way. These can include:

1) **high-humidity chambers** – in which steam is injected providing a controlled humidity;
2) **high-temperature chambers** – thermostatically controlled ovens with internal fans to maintain even temperature distribution;
3) **low-temperature chambers** – thermostatically controlled refrigerating system used to cool the internal temperature of the chamber;
4) **vibration machine**;
5) **variable air pressure chamber** – from which air may be removed, or pumped in, to maintain a low, or high, air pressure around the tested system.

Above: automatic test equipment. The Series 700, System 720 from MEMBRAIN Schlumberger. (Photo: MEMBRAIN Schlumberger).

Glossary

accelerated failure rate	artificially raised failure rate, caused when a manufacturer tests a device or system under harsh conditions
availability	the period of time a device or system is operational
bathtub curve	failure rate against time curve of a device or system
catastophic failure	failure which is both sudden and complete
complete failure	failure which causes a complete breakdown of a device or system
degradation failure	a failure which is both gradual and partial
failure mechanism	the fault which causes a device or system to breakdown
failure rate, λ	the number of failures occurring in unit time. Normally expressed as a percentage in 1000 hours
gradual failure	failure which is possible to anticipate by examination
inherent weakness failure	failure which is caused by a weakness inherent within a device or system when subject to a stress within its rated capability
maintainability	time taken to repair a failed system
misuse failure	failure caused by the application of stresses beyond the stated capabilities of a device or system
partial failure	failure resulting in a deviation from specified limits, not causing a complete breakdown
reliability	probability of survival
sudden failure	failure which is not possible to anticipate by examination
unreliability	probability of failure

COMPUTERS & SOCIETY

Artificial intelligence-1

What is AI?

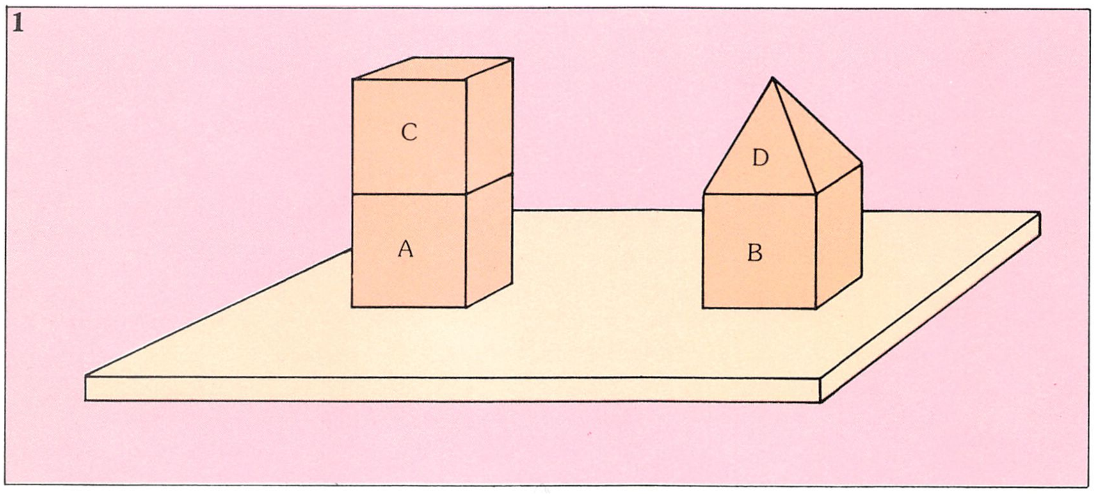
What is artificial intelligence and what type of problem can it solve? Are computers going to take over the world and eventually 'keep us as pets' as one expert would have us believe? These are just some of the questions which people ask.

In these two articles we will try to provide an answer. This first article provides a general overview of the basic techniques used in the field, while the second will examine more complex systems, and attempt to indicate where research is leading.

Artificial Intelligence (AI) is difficult to define in exact terms, but one possible definition is: 'AI is the study of ideas which enable computers to be intelligent'. This is not really satisfactory because it begs the question 'what is intelligence?'. Let us hope that this will become clearer as we proceed. The objective is to extend the range of problems which can be solved with computers to include those which require intelligent decisions to be made.

An example might be to analyse (in real time) the image obtained from a television camera which shows components which have to be assembled, and to move a robot arm to perform the assembly. This is a very complex task involving as it does the identification of the separate components (which may be presented in a variety of orientations); the selection of a rational order to pick up the components (it would be pointless to pick up a nut to put onto a bolt if a washer had not yet been fitted); appropriate movement of the robot arm (some components are taller than others and the 'hand' has to travel high enough to clear these); movement of the hand to grasp the component; and finally, the actual task of fitting the component.

1



1. Robot assembly table-block A has to be put onto block B.

In general, there are two reasons for studying AI. The first is that useful tasks can be performed – suppose that the above system was implemented, then it could be used in places where humans cannot go, inside a nuclear reactor for example. The second is that the algorithms which are used to enable a machine to make rational decisions may shed light upon the way that the human brain reaches its conclusions.

The first of these categories is of interest to software or **knowledge engineers**, while the second is the province of psychologists. We will constrain ourselves mainly to the former category and take a practical approach.

Moving blocks

Since we have started thinking about making a robot assemble components, let us pursue this further and consider how the robot could be programmed to move one block and place it on top of another on the table shown in *figure 1*. For example, to put block A on top of block B, an intelligent human would probably follow the sequence:

1) grasp D and move it out of the way;
2) ungrasp D;
3) grasp C and move it somewhere else;
4) ungrasp C;
5) grasp A and move it onto B;
6) ungrasp A.

The problem is, how do we make a computer make these intelligent decisions, e.g. that C and D have to be moved before A can be put onto D?

Suppose that there are a series of routines available which can do various things with the robot arm:

- PUTON – arranges to put one block on top of another, all action starts with a call to this routine.
- PUTAT – places a block at a specific place specified by co-ordinates, it grasps the block, moves the arm, and then puts the block down.
- GRASP and UNGRASP – make the robot grasp a block and release it. If a block is already grasped when GRASP is called, the block must be put down somewhere, and if the block that is to be grasped is covered, then it must be uncovered.
- CLEARTOP – clears the top of a block however many blocks may cover it.
- GETRIDOF – moves a single block from the top of another and puts it somewhere on the table.
- GETSPACE – makes space on the top of the target block.
- FINDSPACE – finds out if there is space on the target block, if there is not then it gives up.
- MAKESPACE – is cleverer than FIND-SPACE and removes the blocks which are in the way.
- MOVE – moves the arm to the specified position.

Given these routines we can see how the system can move A onto B. Firstly we ask PUTON to put A onto B. PUTON asks GETSPACE to make space on B. GETSPACE asks FINDSPACE whether the top of B is free. FINDSPACE says 'no', so GETSPACE asks MAKESPACE to make the top of B free. MAKESPACE asks GETRIDOF to put D somewhere else, and to do this GETRIDOF asks FINDSPACE to get it a co-ordinate on the table which is not occupied, and then asks PUTAT to do the actual move. PUTAT asks GRASP to pick D up, MOVE to go to the co-ordinate given, and UNGRASP to put the object down. The routines then return up to PUTON which now knows that the top of B is free.

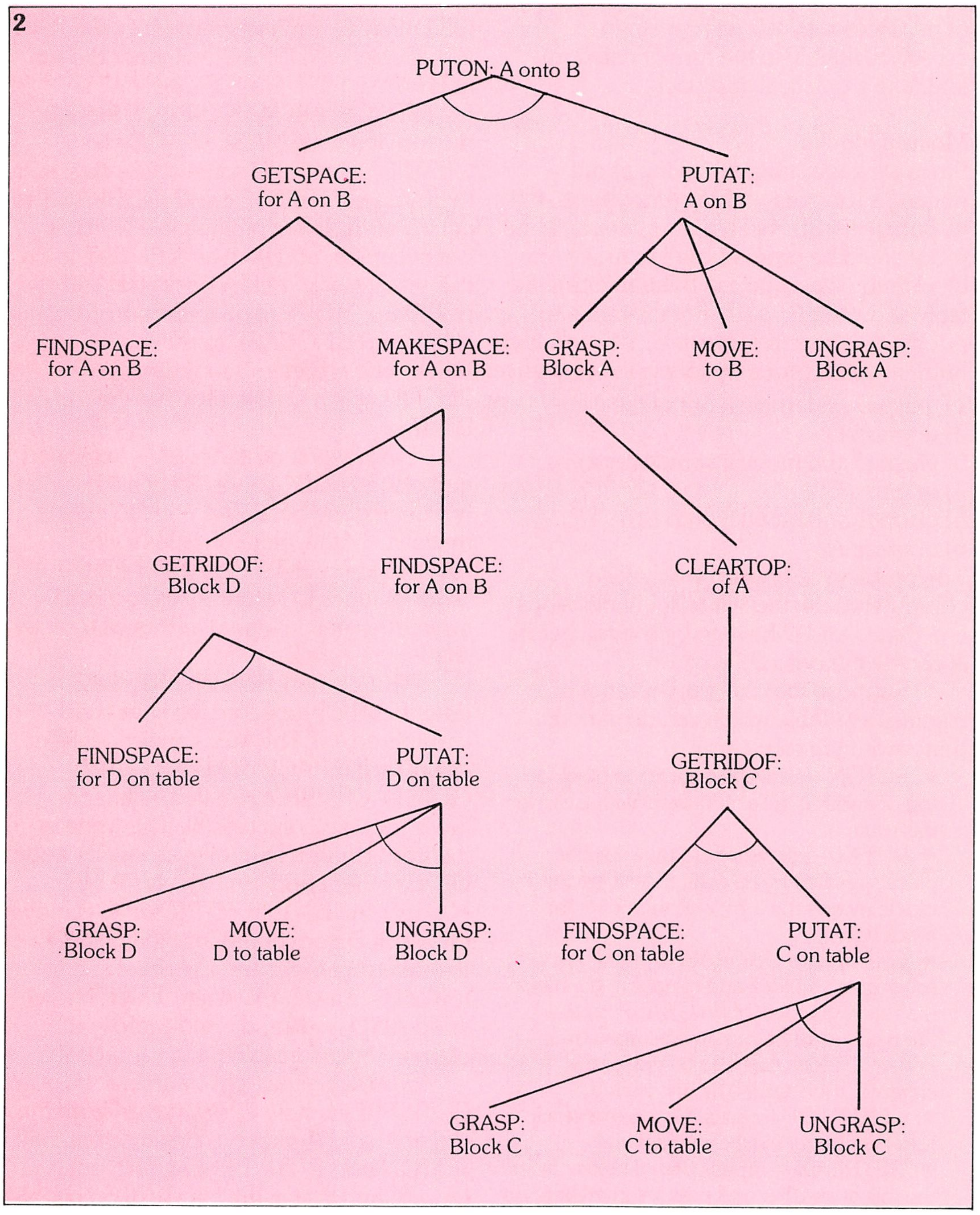
Before looking at how A is extricated and placed on B, we should consider what has been done here. The point to note is that each of the **specialists**, as such routines are called, made **goals** for the other routines to follow. These goals can be represented in a **goal/sub-goal tree**, as shown in *figure 2*.

The left-hand branch of *figure 2* describes the procedure we have just considered. GETSPACE can set goals for either FINDSPACE or MAKESPACE. GETSPACE will be satisfied if either of these routines is successful. This node in the tree is therefore known as an **OR node**. On the other hand, both goals set by MAKESPACE – for GETRIDOF to remove the block D from B and for FINDSPACE to make sure there are no further blocks on top of B – must be satisfied. This is known as an **AND node** and is indicated in the diagram by an arc joining the sub-goals.

Following the right-hand branch from PUTAT to see how C is removed from the top of A, and A put onto B, we notice that PUTON asks PUTAT first of all to put A onto B, and that in the course of doing this another task is generated for PUTON – to move C from the top of A into space on the table. The routine is therefore **recursive** since it can call itself. This process of generating and executing the goal tree is known as **goal reduction**.

One interesting feature of programs using the goal tree structure, which is, in fact, a general characteristic of most practical AI systems, is their ability to answer questions about why they have made certain decisions. Suppose that the system has managed to put A onto B, and

2

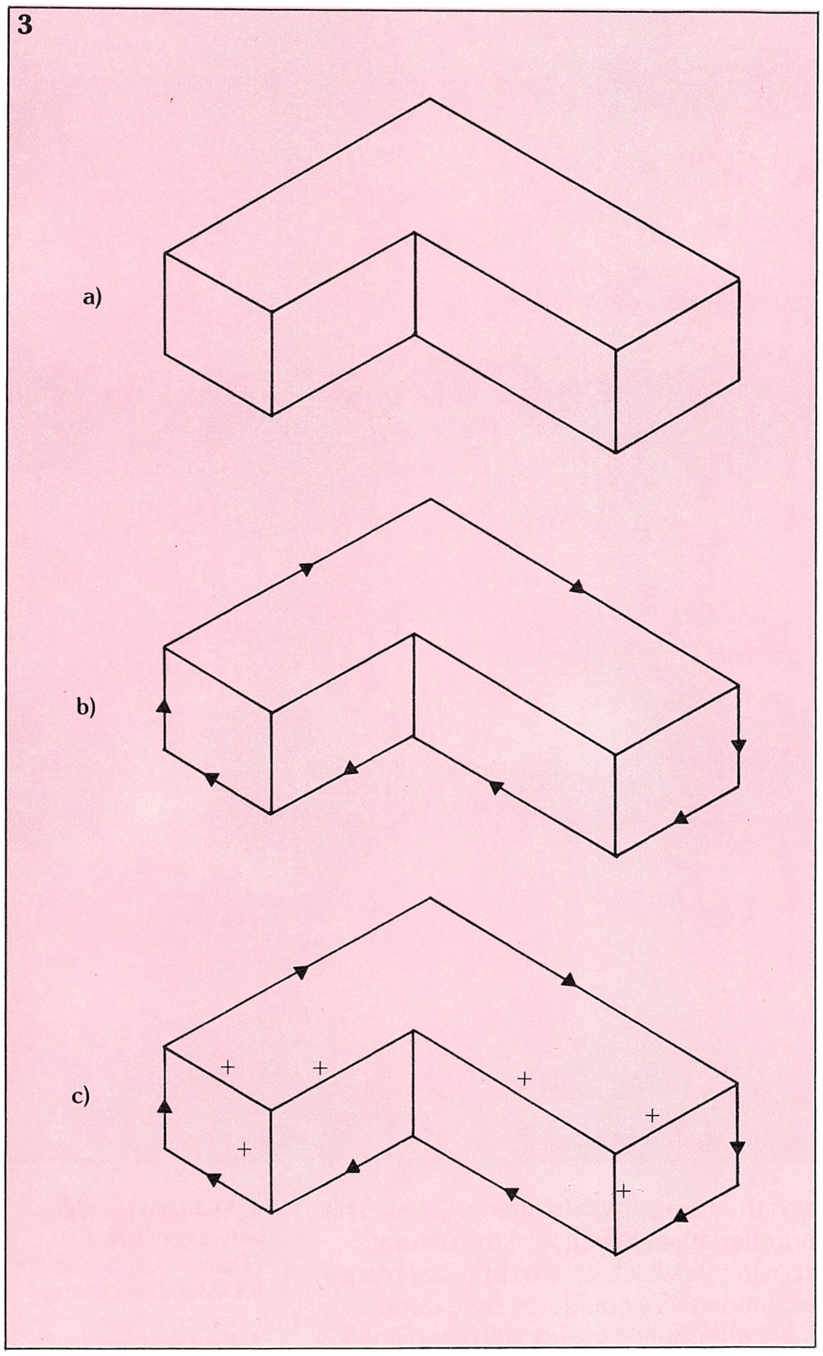


2. Goal/sub-goal tree showing how block A is moved onto block B.

someone asks 'how did you clear the top of A?' Searching the right path of the goal tree gets us to CLEARTOP A. Then to answer the question is trivial, 'by getting rid of C', i.e. the sub-goal that it set for GETRIDOF. Alternatively, if we ask '*why* did you clear the top of A?' then the answer comes back, 'so that I could grasp A', i.e. the goal that it was set by GRASP. We will see examples of more sophisticated decision justification later in the article.

Natural constraints

Until now it has been naively assumed that the computer system knew exactly the position of each block, its orientation, and whether or not it was covered by other blocks. One way that a system could gain this information is to ask a human viewer, but this would involve a lengthy dialogue through a keyboard. There is, therefore, much interest in the fields of **picture analysis** and **natural language** understanding.



3. Line labelling of a solid L-block.

Picture analysis concerns the conversion of a scene viewed through a TV camera into relationships between recognisable objects. Natural language is the sort of language that we all use – full of grammatical errors, many different ways of saying the same thing, many colloquialisms. Understanding such speech or typed phrases is a very difficult problem in AI, and until it is adequately solved it will not be possible to have the type of conversations with our computers that the astronaut has with HAL in the film 2001: A Space Odyssey.

The basic method of attack in these two cases rests upon the exploitation of **natural constraints**. Consider human interpretation of spoken text. The interpretation does not just depend upon the meaning of the individual words. Some words are ambiguous and their meaning has to be worked out from their context. Sometimes the meaning is altered simply by the way in which the phrase is spoken or the way in which a different emphasis is given. Even the problem of understanding written text is very difficult. One way of approaching the problem is to study natural constraints in the way that we use language. If some words have been understood, then there is often only a limited number of interpretations for the remainder which is consistent with what is already known.

Picture analysis

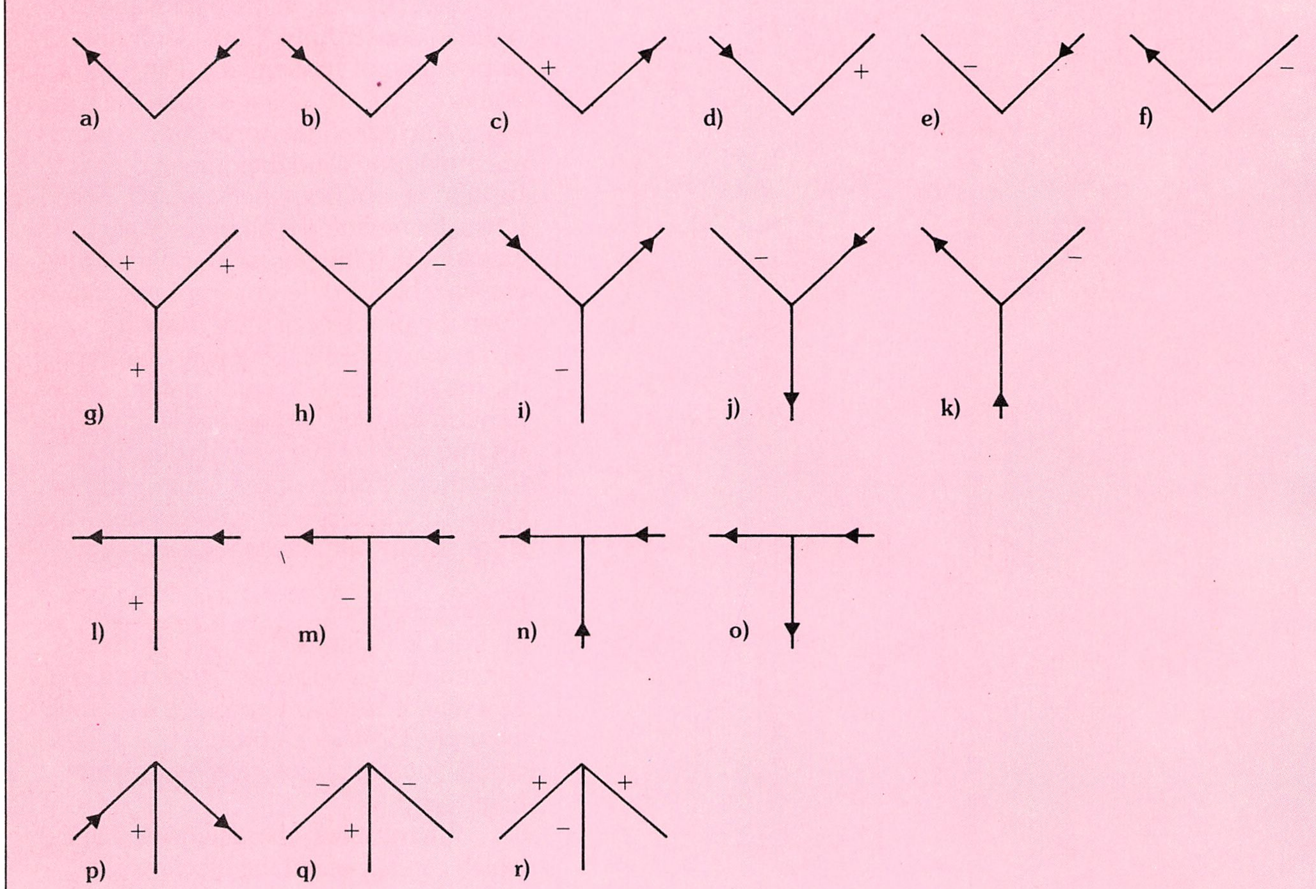
In order to illustrate the use of natural constraints, we will consider picture analysis where it is easier to provide a realistic example. Consider a picture of the 'block world' such as our block moving robot might see.

We must first characterise the lines which can appear in the block world picture. Suppose that we have the L-shaped solid of *figure 3a*. Most of us will 'see' this as an L block lying on top of a surface. The alternative way of seeing the lines is as the ceiling of a room with the picture rail running around. This second interpretation takes some effort to see, and does not represent the outside of a solid block but the inside. We shall only concern ourselves here with the outside of blocks.

Assume a simple situation where:
1) the blocks cast no shadows and have no cracks in them;
2) all vertices of the blocks are the intersection of just three faces (i.e. no complicated blocks);
3) we are viewing from a position where if we move slightly, the junctions between the faces do not change type.
The usefulness of these restrictions will soon become apparent.

Labelling the lines exposes the constraints which enable the interpretation to be made: walk around the extreme **bound-**

4



ary lines keeping the body of the object on the right and mark the lines which are travelled with the direction. We then have the lines marked as in *figure 3b*. The remaining **interior** lines are labelled according to whether they lie at the junction of **convex** surfaces (these are labelled '+'), or **concave** surfaces (which are labelled '–'), as in *figure 3c*.

This provides us with a range of possible **junction configurations**. These are the different ways in which the three lines which form a vertex and the angles between the lines (whether acute or obtuse) can be labelled. Combinatorially there are 208 possible configurations, but in fact, only eighteen occur in real images (given our conditions on viewpoint, etc.) and they are shown in *figures 4a* – r. This number arises because of the natural constraints which govern physical representations. We could not, for example, have a three-line vertex with each line labelled as a boundary.

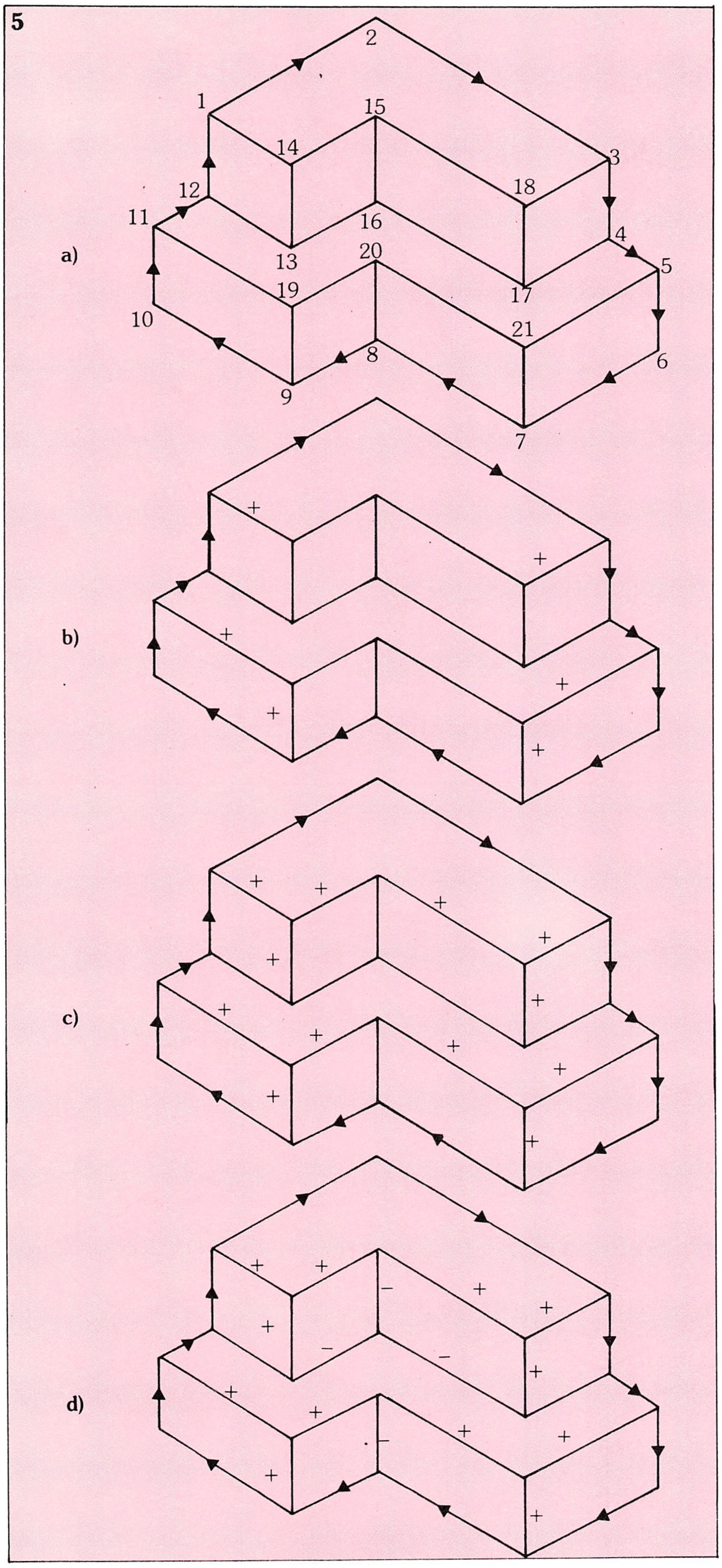
Now, the trick in making the computer understand what it is looking at, is to turn this labelling procedure around: give the computer a list of the possible junctions; have it label the boundary of the object; and then, by a process of elimination, propagate the constraints until there is but one legal labelling.

To take an example, look at the two-tier L-block of *figure 5a* which has its boundary marked and its junctions numbered 1-21. Given this boundary, we can

4. Eighteen possible vertex configurations. (There would be 208 but for natural constraints).

5. Propagation of junction labels to 'understand' object.

5



examine each of the junctions which lie on the boundary and compare them with those in the set of *figure 4*. *Figure 5b* shows that junctions 1, 3, 5, 7, 9 and 11 match only junction type *p* from the available set. The only configuration which includes only internal lines separated by obtuse angles with one line labelled '+' is *g*. Both of the other lines in this configuration are also labelled '+', so we can **propagate** these constraints to the inner lines through junctions 14, 18, 19 and 21 as shown in *figure 5c*. These deduced line labels can now be used in selecting other configurations for adjacent junctions. You should be able to finish the labelling as shown in *figure 5d* by using the junction types *r* and *h*.

So, by propagating what we know about the natural configurations of these junctions we have generated the one legal labelling which shows how the lines are to be interpreted as a three dimensional object. If an object such as that shown in *figure 6* is used, then the last junction cannot be labelled because it is not in the set of junctions, and the computer recognises that something is wrong.

This type of natural constraint propagation has been generalised into an algorithm known as **Waltz's Procedure** which permits the 'understanding' of quite complex pictures. In real systems the restrictions on viewpoint, etc., are relaxed, and the size of the set of possible junction configurations – known as **Waltz's Set** – rises to about 1800. This set takes into account the non-general viewpoint, and includes the junctions which arise when illumination and shadowing are allowed, plus cracks (where two objects touch along a line).

Waltz's procedure operates by listing, for each junction, the possibilities from the available set, and as the constraints propagate, it reduces the possibilities which are consistent with other junctions sharing common lines.

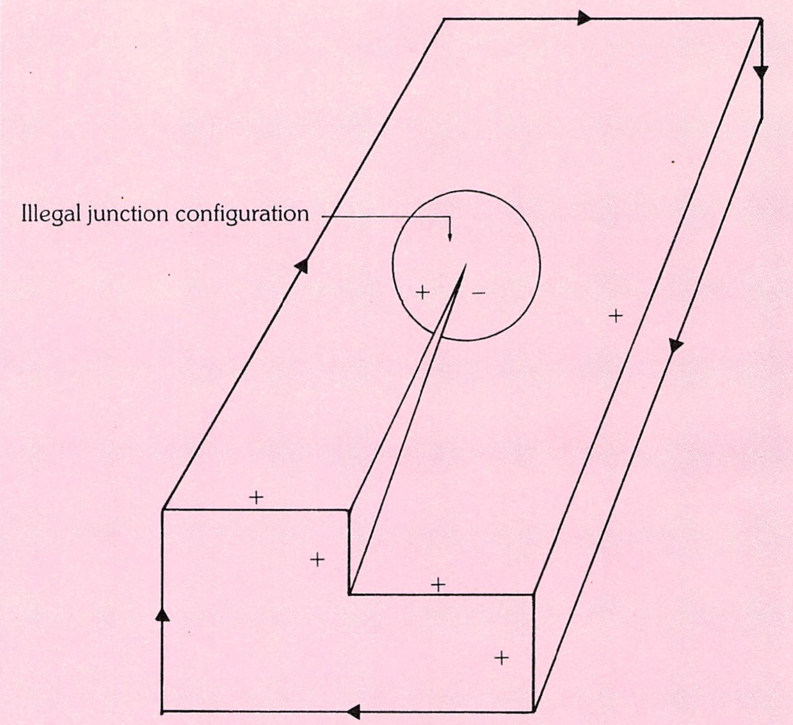
There is evidence that the human brain operates upon a similar principle. Note that our label propagation method relies upon correctly distinguishing the boundary of the object. If you cover the boundary lines of *figure 7* with your fingers then you can interpret the central part in a variety of different ways.

The exploration of alternatives

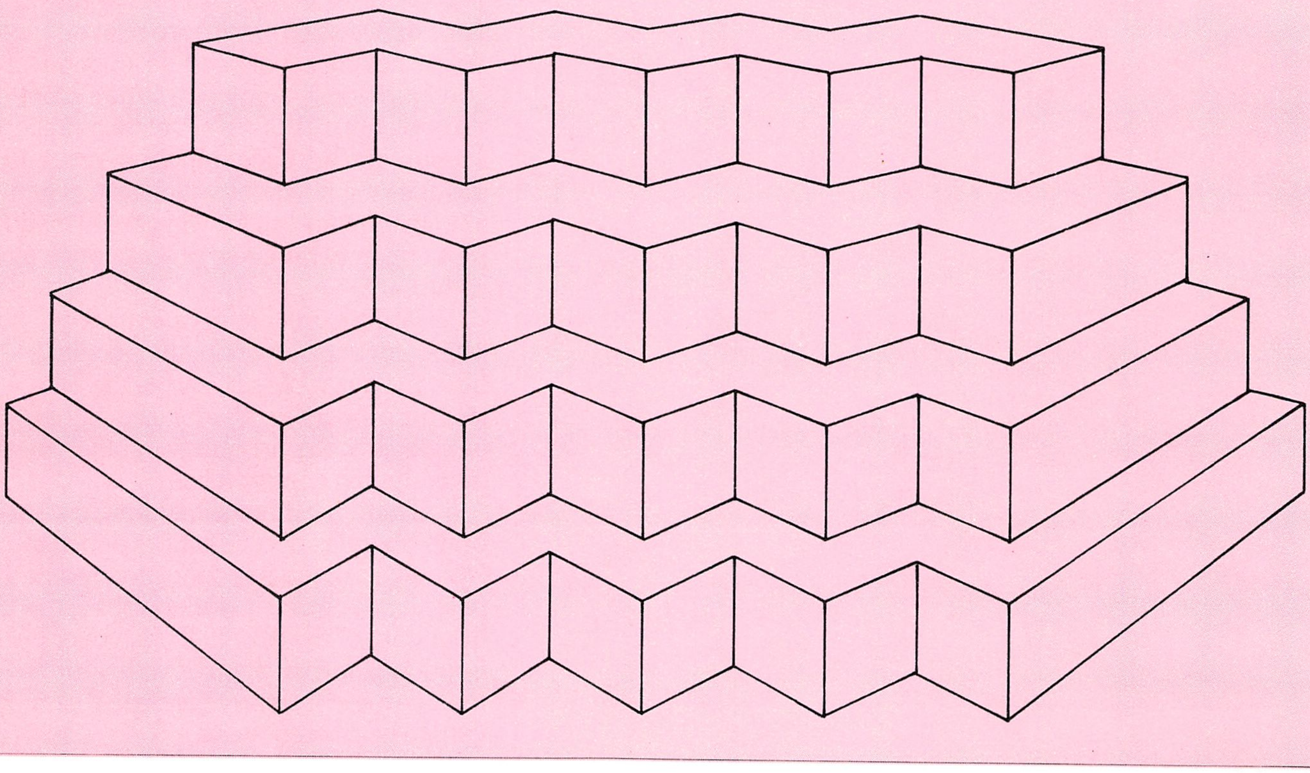
Now consider a different topic: how do we explore a range of different possible options to try to assess the longer term effects of selecting them? Each choice leads to another, and each choice obviously has some effect – it is a good choice or a bad choice. This is essentially a search problem, and the problem is ubiquitous, occurring in applications as diverse as route finding and game playing. Let us look briefly at several different search strategies by using the example of path finding. If we only needed to find a path for use infrequently then there is little point in spending much effort to find the optimally short path. So we shall initially concern ourselves with just finding any path between two points.

Suppose that we have the net shown in *figure 8a*, and we want to find a path from Edinburgh, the **source**, to Paris, the **goal**, when, because of an air traffic

6



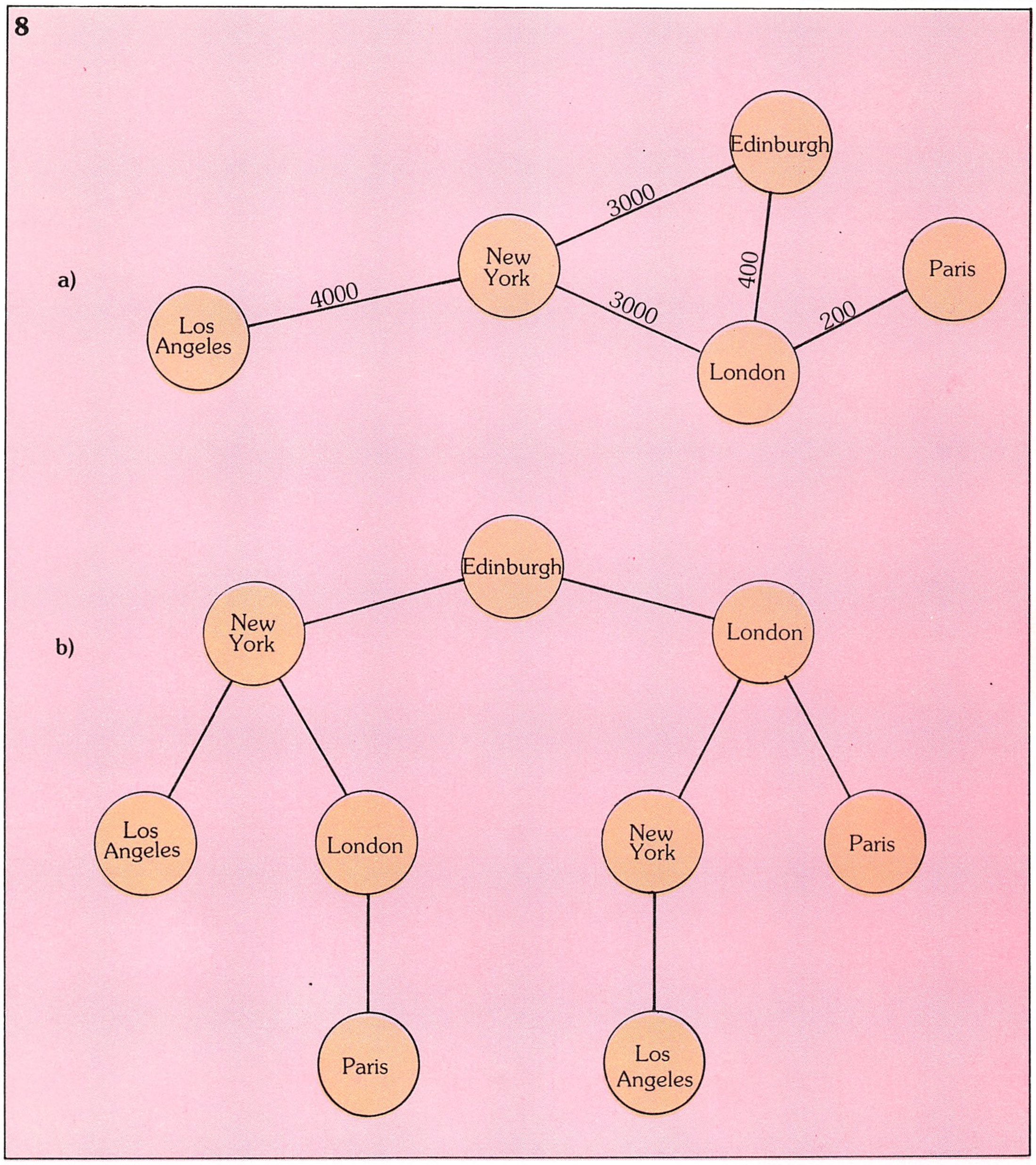
7



6. Propagation of constraints over illegal block reveals error.

7. The brain uses natural constraints – cover the boundary and it is possible to 'see' the central part in a variety of ways.

8



8. (a) Inter-node network; (b) tree representation of network with cycles eliminated.

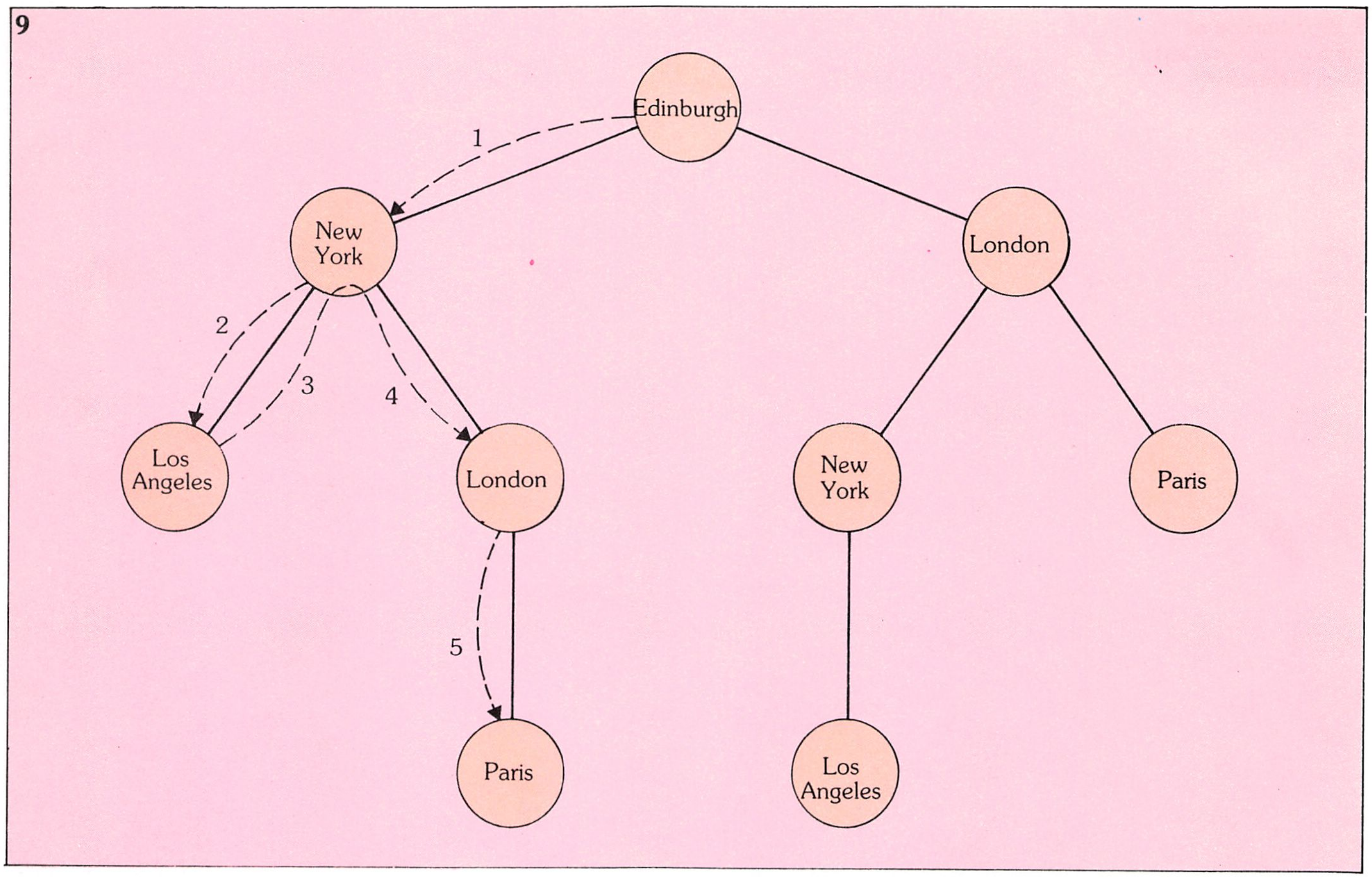
controllers strike, the only available flights are along the links or **edges** between the cities. If we remove cycles from the description (i.e. we would not want to go around the loop Edinburgh-London-New York-Edinburgh-London-New York endlessly), then the net can be transformed into the tree of *figure 8b*.

Depth-first search

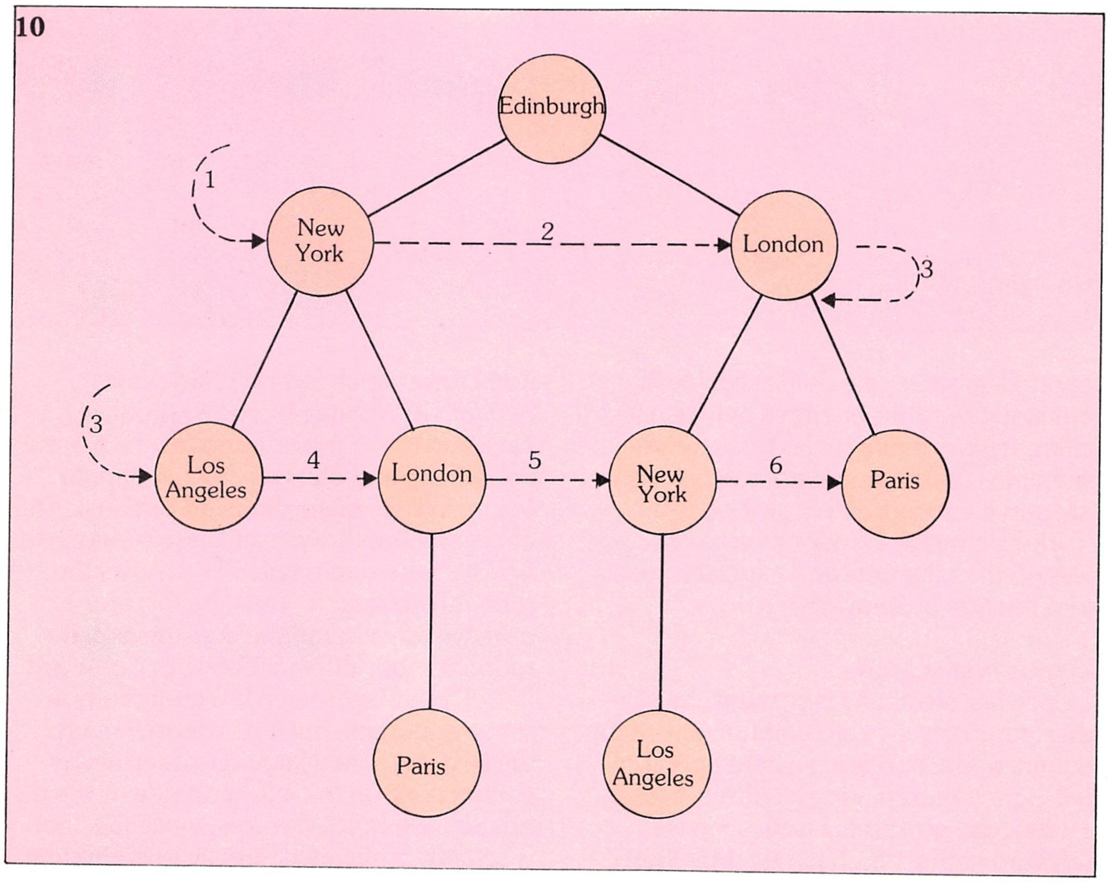
Depth-first searching is perhaps the simplest type of tree search. Starting from the source node, and using a convention of left-right traversal, the algorithm dives as far as it can go into the tree along all leftward paths. When it hits a **terminal node** (one which has no **child nodes** beneath it) without finding the goal, it backs up to the previous node which has a right-child, and plunges down that path (again initially taking leftward options). The effect of this in the present case is shown in *figure 9* where the dotted line shows the route of the search. Thus, by this procedure we have found the goal through the route Edinburgh-New York-London-Paris.

Depth-first search is a good no-nonsense approach, but it has disadvantages: if the tree has a very large and deep set of connections on the left-hand side which do not actually reach the goal, while the goal is actually connected directly to the source

9



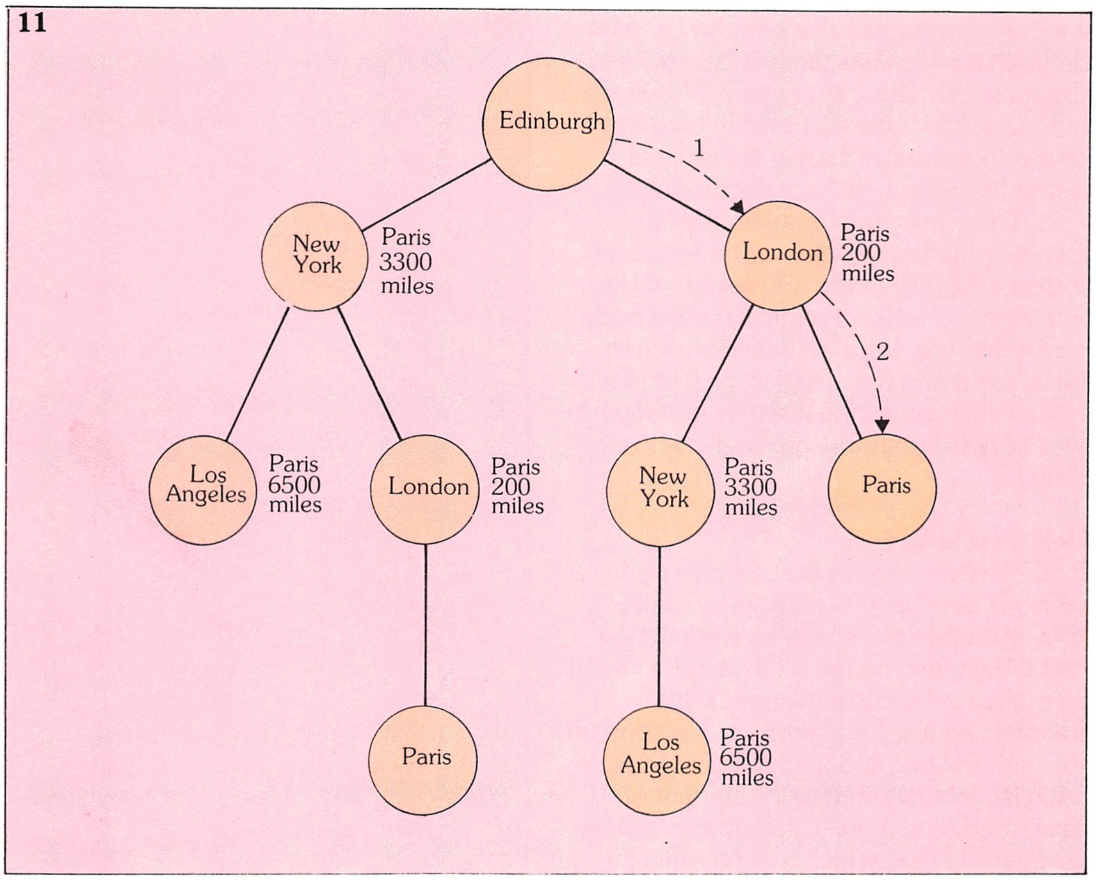
10



9. Depth-first search dives into the tree taking left turns until either forced to back up by a terminal node or until goal is found.

10. Breadth-first search examines all the child nodes at one level before moving to examine their children.

11. Hill climbing dives into the tree, but orders the child node examination by straight line distance from the goal.



as its right-child, then the procedure will search all of the heavy left branch work before trying the right-child and finding the goal.

Breadth-first search
On the principle that if there is a short path from the source to the goal, the goal will lie near the source in the tree, we can use a breadth-first search, which visits all child nodes at the same level of the tree before moving down to the grand-child nodes. This is shown for the example in *figure 10*, where the path Edinburgh-London-Paris has been found earlier than the path found by depth-first search.

This technique also has dangers: if the source is only connected to the goal through many connections, then almost all the nodes in the tree will be visited before a path is located.

Hill climbing
The two simple search procedures, depth-first and breadth-first, can be dramatically improved by the introduction of some method of ordering the way in which the nodes are visited. We will now look at extensions to depth-first searching – first, **hill climbing**.

Imagine that you are halfway up a mountain when a dense fog comes down and that, for some reason, you still have a determination to get to the top. You can stand at any point and measure in which direction the height of the ground around you rises, then take a step in that direction and remeasure the height all around. Then, when the ground falls away on all sides you must be at the top of the mountain.

We can use the same type of movement in the tree of possible routes providing that we have a suitable **metric** which is equivalent to the ground height. One example of such a metric is the straight line distance between where we are now and the goal. *Figure 11* shows some figures which might be available, e.g. London is 200 miles from Paris.

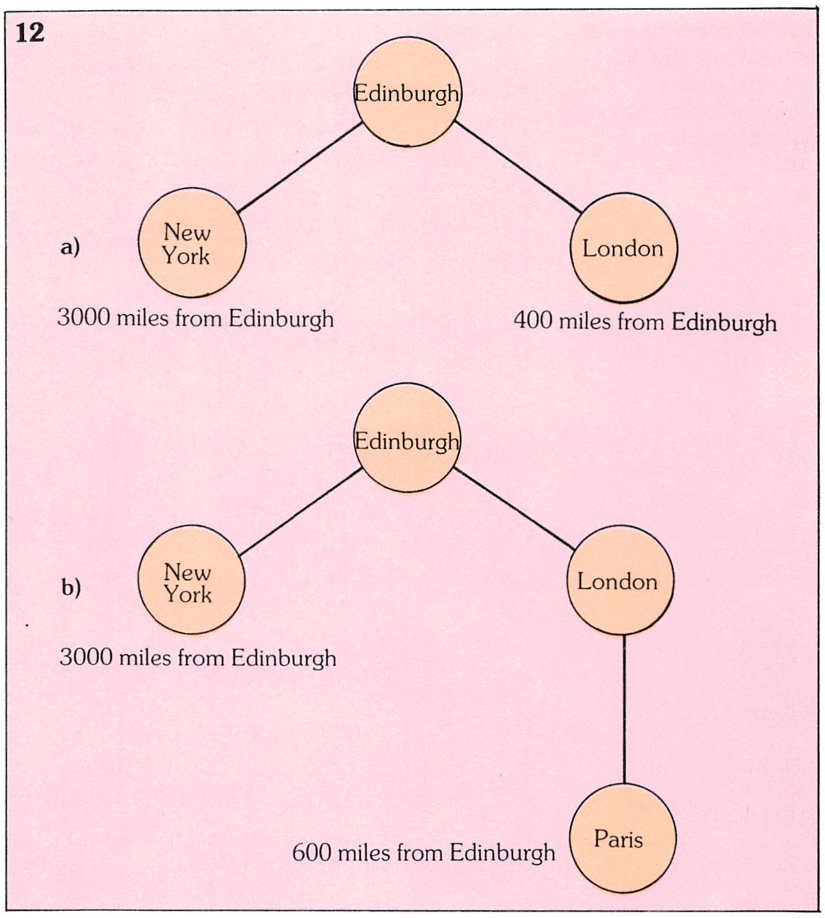
Using basically the same procedure as the depth-first search, we dive into the tree. But this time, instead of always searching the child routes from left to right, we order the paths to be searched, and go down that which has the least straight line distance first. Thus, as in *figure 11*, we are led straight to Paris via London, and the goal is found faster than in the original depth-first search without the distances.

There are problems with the hill climbing method – one may be left standing on a foothill! In the tree search, this is analogous to when a particular child node is selected (say London in preference to New York) and the actual route from that node takes you a long distance out of your way (e.g. Edinburgh-London-New York-Paris).

Best-first search

If hill climbing arrives at a terminal node which is not the goal, it backs up to the previous open node with an unsearched child. Best-first, on the other hand, looks at all of the nodes currently open with unsearched children, and searches forward through the best-looking of them. Thus, at each stage, one more path segment is examined, and then all the open paths are compared to see which is apparently the best route to take.

12



12. Branch-and-bound expands the shortest current path until it knows there can be no shorter paths.

Branch-and-bound search

So far, then, we have been concerned only with finding a path. If however, we have to travel the path frequently, then it might be worth taking rather more trouble and finding the shortest path possible. There are several algorithms used for this, and much work goes into making them efficient. One method is to enumerate all possible paths and choose the shortest. This method, sometimes in jest called the **British Museum method**, utilises the depth-first search with the modification that when the goal is found the procedure does not stop, but backs up and tries other paths. In practice, though, this is only rarely used, since for a problem of significant size it would take a very long time.

One method which is in common use, and which is the foundation for several other methods, is the **branch-and-bound search**. The basic idea of this method is quite simple. Once again we want to travel from Edinburgh to Paris along the network shown in *figure 8*.

Figure 12a shows the first level of the tree, where the numbers this time indicate the distance travelled from the source. Now, if we expand further along the shortest path (from London), we arrive at the goal, in *figure 12b*, after travelling 600 miles. Looking around for other possible paths which might be shorter, we see that the New York route cannot be shorter because we have already used 3000 miles just to get there. So, in branch-and-bound we expand at the end of the shortest path until a path reaches the goal which is of length equal to or shorter than all the incomplete paths.

There is, however, a slight flaw in this explanation. We have not taken into account the distance to get to the goal from the penultimate node. It may be that one of the other open paths is only a very short distance from the goal and that this would make the total distance of that route less than the solution found. To eliminate this problem, we need just a slightly better termination condition. Instead of the above, we terminate when the shortest incomplete path is longer than the shortest complete path.

One interesting method of improving the branch-and-bound search is by the addition of underestimates of the remaining distance. If we make a guess about the distance remaining from any point to the goal, then the sum of this guess and the distance that we have already travelled is an estimate of the total length of that path. We can then work at this path until the estimate is revised upwards sufficiently to make another path estimate shorter. However, guesses are often wrong, and a bad guess can take us a long way out of the best route.

On the other hand, an underestimate of the remaining distance never takes us off the track: if a path to the goal is found by extending the path with the shortest underestimate, then the solution is complete if all incomplete path distance estimates are greater than the completed path. This is true because the other underestimates are guaranteed to be shorter than any real path length through that incomplete route. In the route-finding example that we have been using, the straight line distance remaining is guaranteed to be an underestimate (since no other route can be shorter).

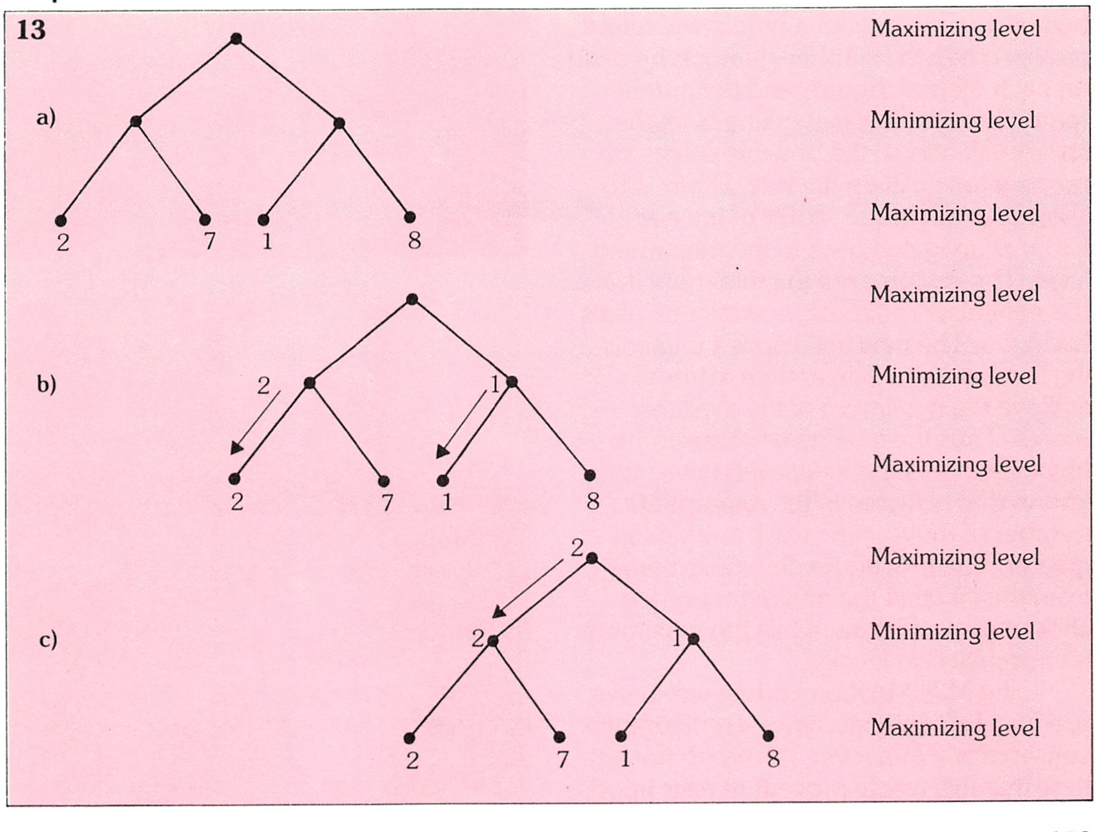
Adversarial games

One of the most widely publicised areas of AI is that dealing with intellectual games, such as draughts or chess. A computer has, in fact, beaten the world draughts champion! It should be obvious that, for game playing to be successful, the computer has to consider the *value* of a move at the current instant in terms of its benefits later on – when the opponent may have made some brilliant move. This is a type of search procedure, where we want to search through a range of possible moves which could be made now and in the future and to select the best one. However, it is not an example of the kind of search with which we have previously been dealing, so a different approach is needed.

Suppose, for the purpose of this discussion, that there is an algorithm which can grade the current state of a chess or draughts board, i.e. it is able to put a number to the quality of the current position. If we are in a good position, then the number is positive, if indeterminate, it is near zero, and if things are looking black then it will be negative. Such an algorithm is termed a **static evaluator**.

13. MINIMAX propagation of maximised/minimised scores on an adversarial game tree.

13



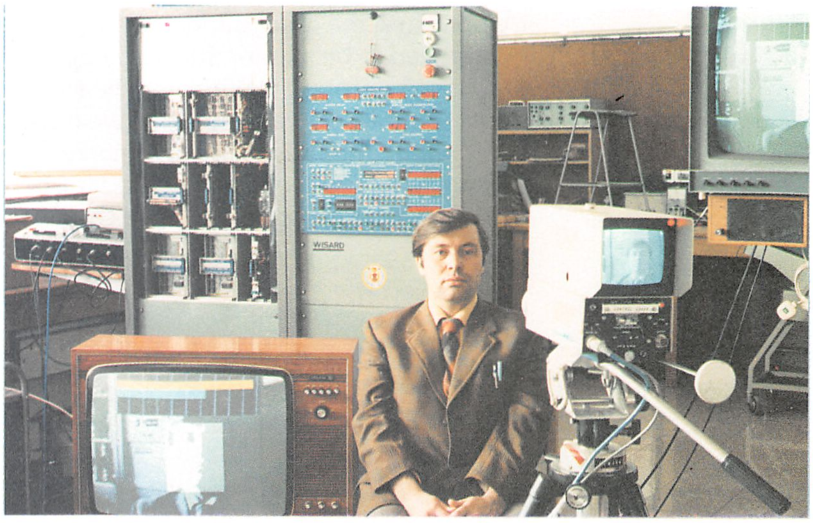
When it is the computer's turn to move, it needs to maximise the benefit by playing the best move. However, if this results in the opponent being able to make an extremely good move which loses the computer the game, then maybe it's original move was not so good. Therefore, as the computer generates new board situations to assess, it needs to look for high positive scores on its moves, but at the same time, it must only choose those moves which limit the possible damage that the opponent can inflict. This can be achieved by using the **MINIMAX procedure**.

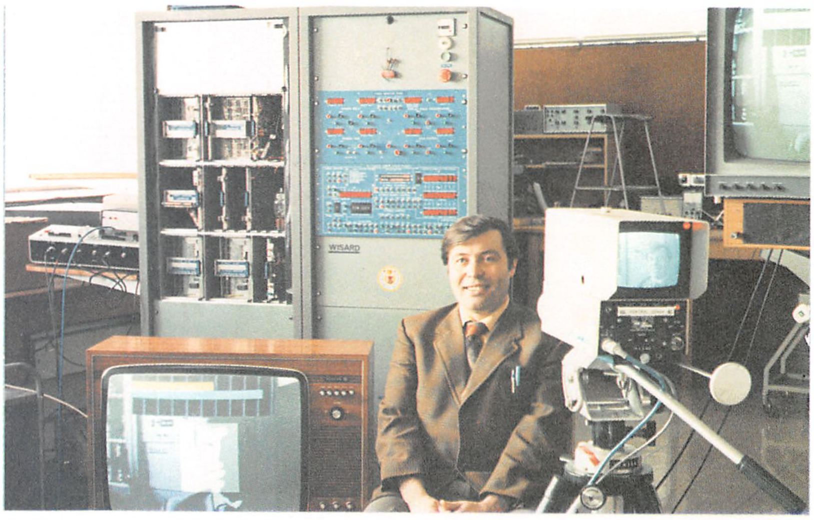
The player who is looking for positive scores is called the **maximiser**; his opponent is the **minimiser**. In the miniature game tree of *figure 13a*, the maximiser might hope to get to the high positive score of eight, two moves in the future. However, he is not very likely to get there because the minimiser could deflect the maximiser's position towards the rather lower score of one. It would be sensible, therefore, for the maximiser to play the other available move, where the maximum possible score is only seven, but where the minimiser cannot reduce this score below two.

The way that this procedure is implemented is to enumerate moves and build the game tree for a number of future moves. Then the static evaluator is invoked on each of these boards and the number remembered. Finally, working from the static evaluations, the possible scores are propagated back up the tree. *Figure 13b* shows the minimum scores of the evaluator propagated back to the minimising level. These scores are the minimum that the minimiser can force, however he plays his move. The next level up is a maximising level, and the player here wants to achieve the maximum of the available scores. Thus the maximiser chooses the highest score at the next level down, and knows that his score in the appropriate number of moves time will be either this or greater. The procedure derives its name from the fact that the minimum score is chosen at even levels, while the maximum is chosen at odd levels.

The MINIMAX procedure will never produce brilliant play, since it is inherently conservative. However, it is worth noting here that this whole procedure rests upon the value produced by the static evaluator. The translation of the 'goodness' of a board into a single number is very difficult, because a single number is not a good summary of the type of information involved. It is also worth noting that the number of possible moves for a complicated game like chess can be very large. There are therefore modifications to the MINIMAX procedure which, rather like branch-and-bound and its derivatives, attempt to limit the number of evaluations which have to be made. Also, since competitive chess and draughts are played with a time limit on each move, breadth-first type strategies are used so that the computer can look as far ahead as is possible in the time available, rather than diving down one possible route and not completing the assessment of the available moves.

Below: the Brunel University WISARD system learns to recognise difficult images. In these two photographs, the system distinguishes between a serious and a cheerful expression. The response is shown on the monitor as a yellow bar for serious and a blue bar for cheerful. (Photo: Brunel University).





Control metaphors

Until now we have said 'X *asks* Y to do Z', as though the procedures were human and listening out for requests. It is useful to think about the control of specialist procedures as though they were human, since the exercise of control is an intelligent process in itself. To solve the complex problems posed by decision making situations we need to have a model of who tells who what to do. This model is called a **control metaphor**. There are several such metaphors which are used to control the way that tasks are assigned in real systems.

In our robotic example, it was not necessary to have particularly clever control since there was only one arm to worry about. If, instead, the robot had been modelled on the human pattern and given two arms, then we would want the two arms to co-operate and to do different jobs at the same time. This is termed a **concurrent** system and the order of complexity is much higher.

Firstly, if both arms are idle and a task needs to be done, then we must select which arm to allocate the job. The choice may depend upon the physical proximity of the task to either arm, or on the fact that one arm is better fitted to perform the task – is the system right or left-handed? We also have to take care that the two tasks do not conflict with each other: is one arm trying to move an object to the right which the other arm is trying to move to the left? Thirdly, we need to ensure that the arms do not attempt to occupy the same point in space at the same time!

The question which underlies the design of a control technique is 'where is knowledge about procedures stored?'. In a conventional **single-thread** program, as produced by a conventional compiler, the programmer knows which routines do which tasks, and calls them accordingly. The knowledge about routines is stored in the calling program. This is known as **action-centred control**, where each specialist knows who to ask to do a particular task.

A second method of controlling the use of routines is to attach the names of suitable routines to the description of the objects for functions. For example, in our two arm system, suppose one arm cannot pick up as heavy objects as the other arm. We can attach the procedure (arm) to be used to the weight description of the object. This is known as **object-centred control**, and is akin to reading the weight description on a bag of potatoes before deciding whether to lift it with left, right or both hands.

There is a third case where the procedures themselves know what they can do. Imagine a blackboard on which the controlling program can write 'I want to lift a 56lb bag of potatoes'. Then, if the procedures are looking, only those which are capable of lifting the bag will respond. This is known as **request-centred control**, and can be likened to a committee of strangers allocating jobs amongst themselves.

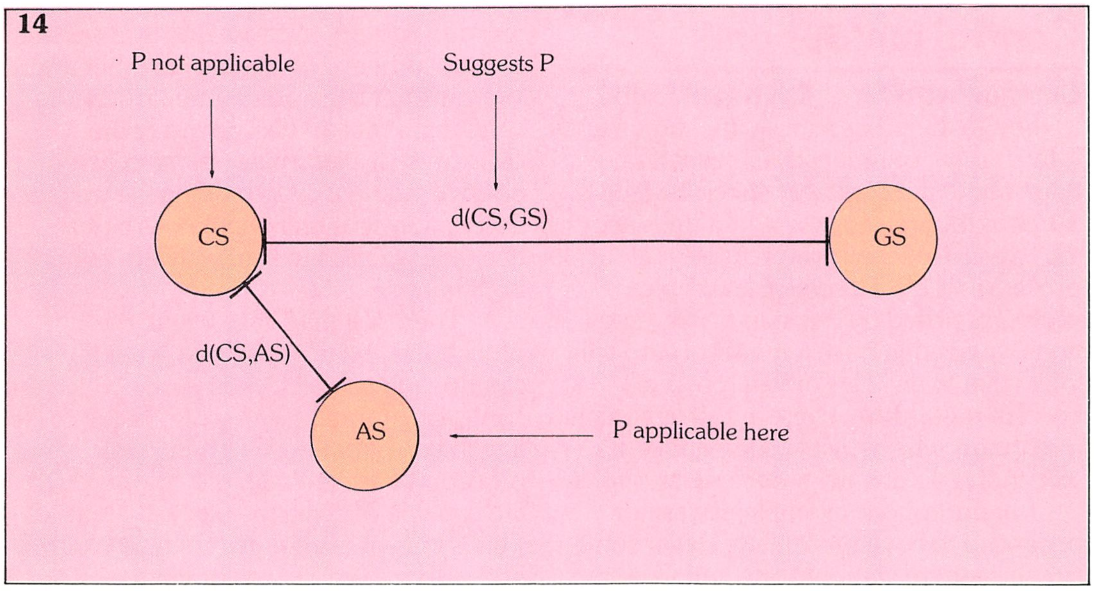
With any type of request-centred control, more than one procedure can respond to a request. Someone has to decide to which procedure to assign the task. Of several approaches, the simplest is to select randomly, and the best is to have the work-requester select the procedure which has the best credentials. We can therefore envisage a situation where the requester converses with the specialists via the blackboard to find which is most suitable. This type of control has been implemented in several practical systems.

General problem solver

Now let us look in more detail at a particular control metaphor, the **General Problem Solver**, GPS. GPS is an action-centred control metaphor – initially developed in the 1950s it has therefore had a long and distinguished existence.

We have developed ideas about how to pick a route between two points in a network. Unfortunately, real life is not quite as simple as the situation presented. If we lived in Edinburgh and were actually going to travel to Paris (and this time we will assume the traffic controllers strike is over and we can get a direct flight), then we would have to *walk* to our car, *drive* to the airport, *walk* to the plane, *fly* to Paris, take a *taxi* to our final destination, and *walk* into the building. This takes us to a new level of detail which we have not yet considered, but these are the things that

14



14. The General Problem Solver creates a subproblem when a suggested procedure, P, cannot be applied to the current state, CS, but can be applied to a nearby adjacent state, AS.

15. Difference-procedure table which enables GPS to choose the correct procedure at the correct time.

15

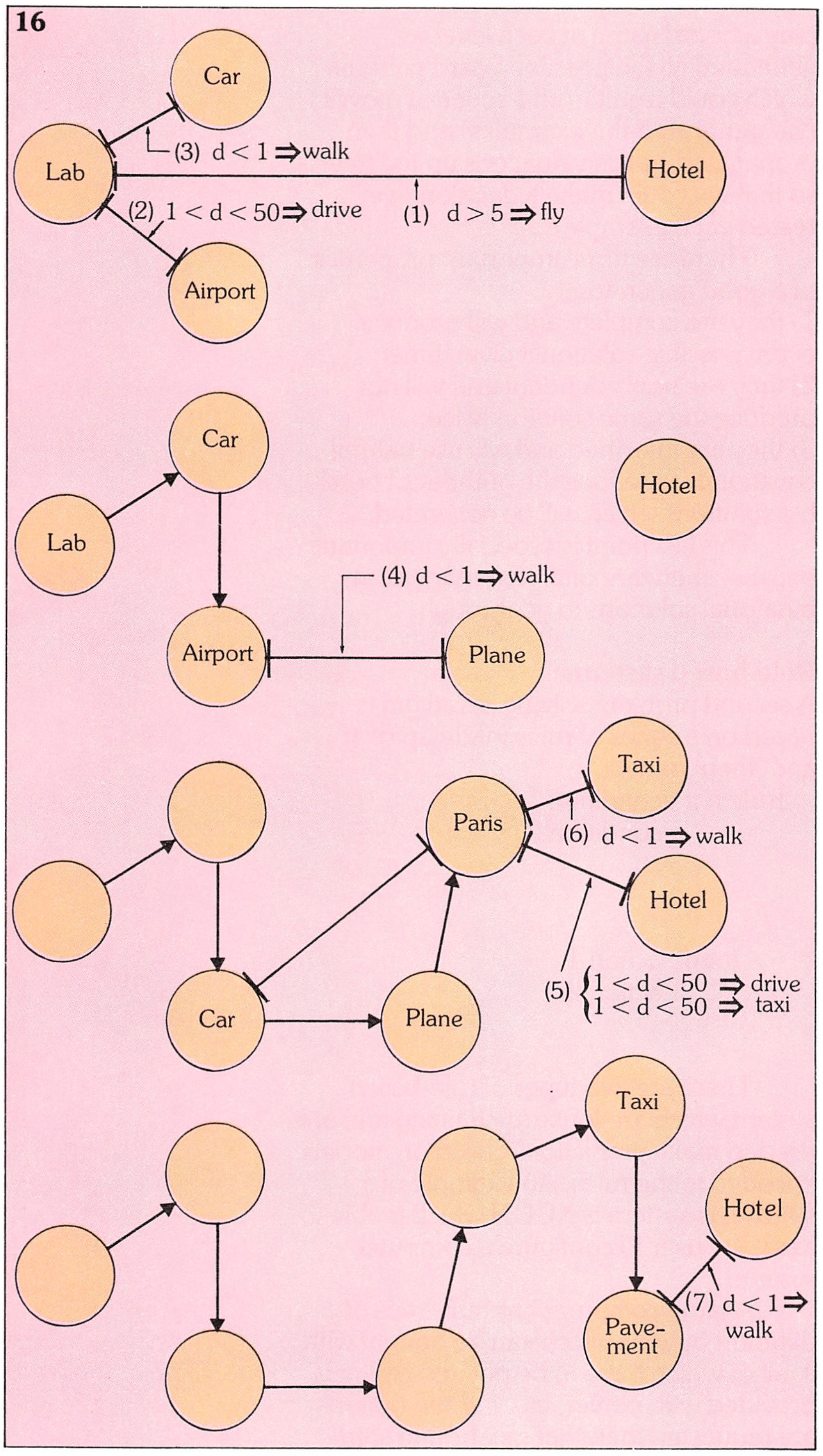
	Procedures			
Distance to goal, d (miles)	Fly	Drive	Taxi	Walk
d > 50	✓			
1 < d < 50		✓	✓	
d < 1				✓
	Be at plane	Be at car	Be at taxi	None
	Prerequisites			

our robot (let's call him Robbie) would have to do if, after a hard year moving blocks around, he was taking his annual holiday in Paris.

Robbie has a list of the means of transport (walking, driving, flying, etc.) and he has to decide when to use them in the context of his journey. These means are metaphors for any of the routines that we discussed for moving blocks around, etc. We put these under the control of GPS which decides when to use each of them. The key idea of GPS is to operate to reduce the difference, d(CS,GS), between the current state, CS, and the goal state, GS. The fact that Paris is several hundred miles and across water from Edinburgh suggests that Robbie should take a plane and fly. But planes do not fly from the laboratory where Robbie works, and so he cannot use the procedure 'fly' from his present position. However, he knows that nearby there is an adjacent state, AS, the airport, where 'fly' is legal. So he recursively calls GPS and asks it to solve the problem of getting him to the airport. These ideas are shown diagrammatically in *figure 14*.

Let us look at how Robbie would set up and solve the problem. Robbie has a table like that shown in *figure 15* telling him which procedure to use depending on the geographical distance between his CS and his GS. The table also contains a list of prerequisites to the use of each procedure, for example, he has to be at a plane before he can fly. This table is known as a **difference-procedure table**.

Figure 16 shows what happens when Robbie makes his journey by asking GPS to get him to his hotel.



16. Robbie's journey to his Paris hotel under the control of GPS.

1) Firstly, he decides that Paris is more than 50 miles away, and that he must therefore fly. He is not at the airport, however, so he asks GPS to get him there.
2) The airport is further away than his walking distance so he must drive. He is not, however at his car, so he passes this problem to GPS also.
3) The car is in the garage about 50 yards away, which is within his walking distance. He has no prerequisites for walking, so at last he is off down the road, executing his 'walk' procedure. Procedures 'drive' and 'fly' remain in wait states. As soon as he reaches the car his 'drive' procedure takes over, and finally he reaches the airport.

4) At this stage, he faces the problem that he cannot yet fly because he is still not at the plane – he is still sitting in his car. So he asks GPS to get him to the plane again, but this time things have changed, and the plane is less than a mile away so he can invoke his 'walk' procedure. This gets him to the plane, when his 'fly' procedure takes over and eventually they land him in Paris.

5) Now he finds himself less than 50 miles from his hotel so he decides to drive – except that his car is back in Edinburgh and would involve him in flying back to fetch it! So, hopefully Robbie's GPS system rejects options which take him to an AS which is farther from his GS than his CS. Instead Robbie chooses the 'taxi' procedure, but to get to the taxi he has to ask GPS again.

6) His distance from the taxi is less than one mile so he 'walks', and when he gets to the rank his 'taxi' procedure operates, and leaves him on the pavement outside the hotel.

7) The hotel is now quite close and GPS decides that Robbie ought to walk the remaining distance.

Thus Robbie has been safely delivered to his hotel by GPS calling the procedures which were appropriate to the particular task.

It is worth noting one or two points about GPS:
1) The representation of the information in the difference-procedure table can be altered to provide suitable mechanisms for other types of problem, and this makes GPS a widely used control metaphor.
2) GPS is hierarchical and recursive – it splits problems down into manageable units, and calls itself to solve these sub-problems.
3) GPS makes no statements about resource allocation, and it does not account for possible concurrency – only one procedure is active at any one time.

Problem solving paradigms

As you have probably realised by now, much of AI is concerned with solving problems in fairly general ways. It is therefore necessary to have a range of models which describe different ways of approaching problems. These models are called **paradigms**. In looking at GPS, we treated it as a control metaphor since it dispensed tasks to particular procedures which could solve the part of the problem which they were given, while GPS worried about the overall problem. GPS can also be viewed as a **problem solving paradigm**.

1) It moves forward iteratively searching for a goal, the search which it uses being depth-first (it dives down to try to solve the very small details before it can back-up and get on with the next part).
2) It creates a goal for each procedure when it is called – thus it is using goal reduction and could answer questions about why it has done various things.
3) It can be modified to defer action on the little details until the overall path is broadly clear, and thus it can include a kind of planning.

Systems are said to be **forward-chaining** if they search towards the goal from an initial condition like GPS, and **backward-chaining** if they work from the goal state towards an initial condition. GPS can be modified to be backward-chaining, but the most suitable depends upon the problem being solved. If solutions tend to **fan-out** from initial conditions to many goal states, then backward chaining is better because a forward-chain could go down many cul-de-sacs as we saw with depth-first searching. However, if many possible initial states **fan-in** to a few goal states, then it is better to use forward chaining since this will lead straight to a goal state.

There are other problem solving paradigms, and we will now look briefly at two others.

Generate-and-test

We have already seen an example of this paradigm. When we used MINIMAX to play a board game at each level we generated all the possible board patterns which could result from a series of moves (we **generated** the examples) and then propagated static evaluations up the tree so that we could make a decision (we **tested** each example).

There are three important properties of a good generator:
1) they are complete and will produce every possible solution if given time;
2) they are nonredundant and will not produce the same solution twice;
3) they are informed and will use natural constraints to reduce the number of possible solutions which will be generated.

This last point is especially important because there are often too many combinatorial solutions to go through.

Rule-based systems

A second problem solving paradigm is based on a series of rules made up of 'if' and 'then' parts:

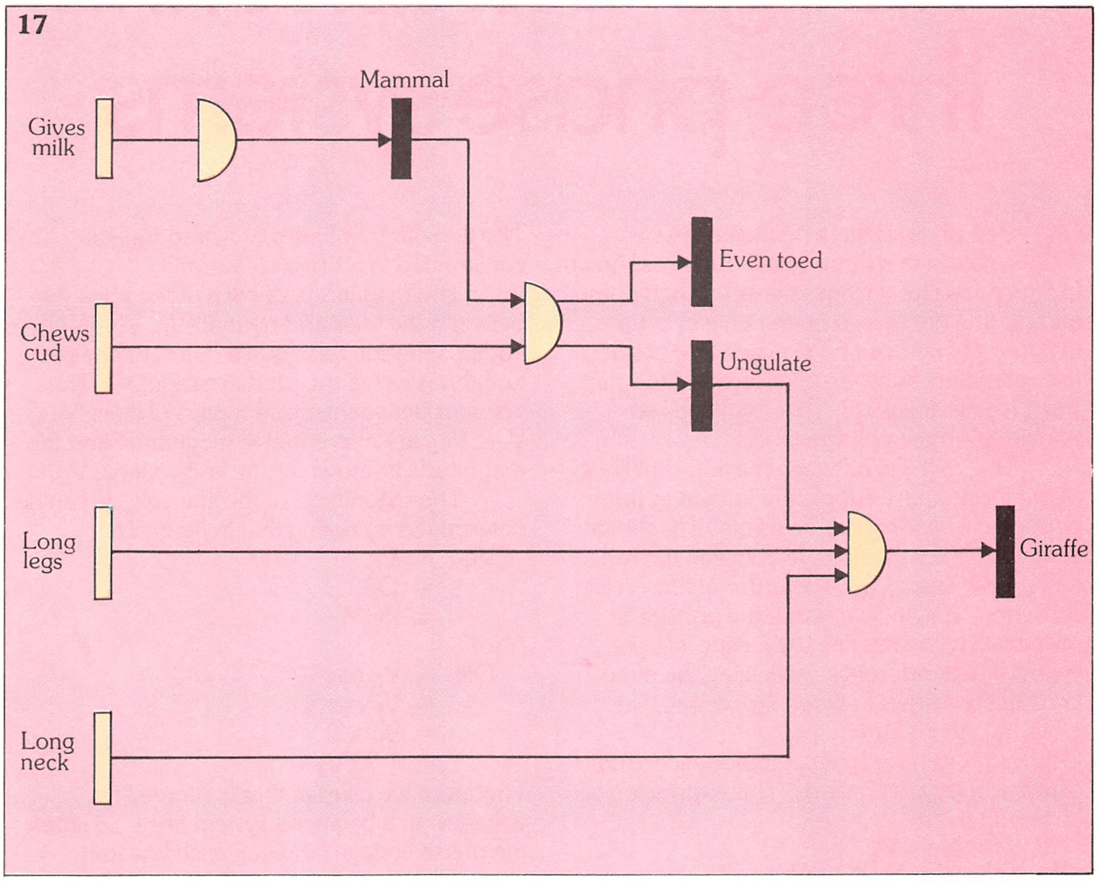
```
Rule n if  condition 1
           condition 2
           .
           .
           .
      then action 1
           action 2
           .
           .
```

There are two types of rule-based system which are forward-chaining and are used to make predictions based on models encoded in the rules. An example of a **synthesis** system is **XCON** which is able to assist the user to configure a computer system.

A large computer has hundreds of different options which can be bought with it, all of which have to be put into cabinets, provided with power, etc. If some options are requested then that means that some others, *must* be present also. This complicated task is known as configuration, and XCON is better at it than most humans.

However, there are other real systems which fall into the **analysis** class of rule-based system: such systems are commonly called **expert systems**. Such systems can either be forward or backward-chaining depending on the application. They are able to make deductions from

17. Inference net describing how the identity of the giraffe is deduced from the four raw facts.



information that is provided by the user, usually interactively through a keyboard. The **rule-base** or **knowledge-base** contained in these systems comprise **antecedent-consequent** rules which can be chained in either direction. All of these systems can answer questions about their 'thinking' by repeating the rule information along the thought-path. Sometimes, an extra part of the rule is included:

```
  Rule n  if        antecedent
          then      consequence
          because   reason text
```

and this can be used to provide more detailed information.

The rules form part of an **inference net**. An example is shown in *figure 17*, which is part of an animal identification system which would allow Robbie to distinguish between different animals by reasoning from facts which he can observe to the animals names. The orange rectangles represent raw facts; the black rectangles are facts which are deducible from the raw information; and the symbols which look like AND gates represent the rules. The inputs to these **inference gates** if all true will **trigger** the rule, and a deduced fact will be output from the consequence part of the rule. So, the known fact that an animal gives milk allows us to deduce that it is a mammal. Because we know that the animal chews the cud, and we have deduced that it is a mammal then we can further deduce that the animal is an ungulate. This deduced fact coming together with the observable facts that the animal has long legs and a long neck permits us to deduce that the mysterious animal is a giraffe.

Conclusion

In the second of these two articles on AI we will look at:

1) antecedent/consequent (expert) systems in more detail:
2) programming in logic (a generalisation of the rule-based systems that we have been discussing);
3) the state of the art – what is happening in AI now;
4) Japanese fifth generation computers;
5) the British Alvey initiative on information technology.

BASIC THEORY REFRESHER

ELECTRICAL TECHNOLOGY

Voltage and current in three-phase systems

A three-phase system connected in star formation to a three-phase supply is shown in *figure 1a*. The voltages between each phase and neutral are shown on the phasor diagram in *figure 1b*, and can be measured by connecting voltmeters between each line and the star point (or neutral wire). These voltages are known as **phase voltages**.

The voltages between the adjacent lines of the three-phase supply are known as **line voltages**. These can be determined as shown in *figure 1b*, for the voltage between the red and yellow lines, V_{RY}. The order of the subscripts, RY, indicates that the voltage is measured at the red line with respect to the yellow line as reference. Following the directions of the arrows in *figure 1a*, we see that:

$$\mathbf{V}_R = \mathbf{V}_Y + \mathbf{V}_{RY}$$

(The voltages have been emboldened to indicate that they are phasors.) This expression can be rewritten as:

$$\mathbf{V}_{RY} = \mathbf{V}_R - \mathbf{V}_Y$$

In order to show this as a phasor diagram, the phasor representing $-V_Y$ (which is opposite to V_Y) must first be constructed. These voltages can then be added together using the parallelogram rule, to give V_{RY}: the voltage on the red line with reference to the yellow line. We can see that V_{RY} leads V_R by an angle of 30° ($\pi/6$). The two other voltages, V_{YB} and V_{BR}, are constructed in a similar manner.

The magnitude of each of the voltages between the lines and neutral (V_R, V_Y, V_B) is equal, although each is at a different phase; and this is called the phase voltage, V_P. The voltages between adjacent pairs of lines, V_{RY}, V_{BR}, V_{YB} are also equal in magnitude and this magnitude is known as the line voltage, V_L.

The magnitude of the line voltage can be determined by noting that in *figure 1b*:

$$\begin{aligned} V_{RY} &= V_L \\ &= OP \\ &= 2OM \end{aligned}$$

Also:

$$\begin{aligned} OM &= V_R \cos 30° \\ &= V_P \cos 30° \\ &= V_P \sqrt{3} \\ &= 1.73\, V_P \end{aligned}$$

Therefore we can see that the three line voltages on a balanced system are 1.73 times the phase voltage between each line and neutral. The phase voltage in the U.K. low voltage distribution system is 250 V, thus the line voltage can be calculated as:

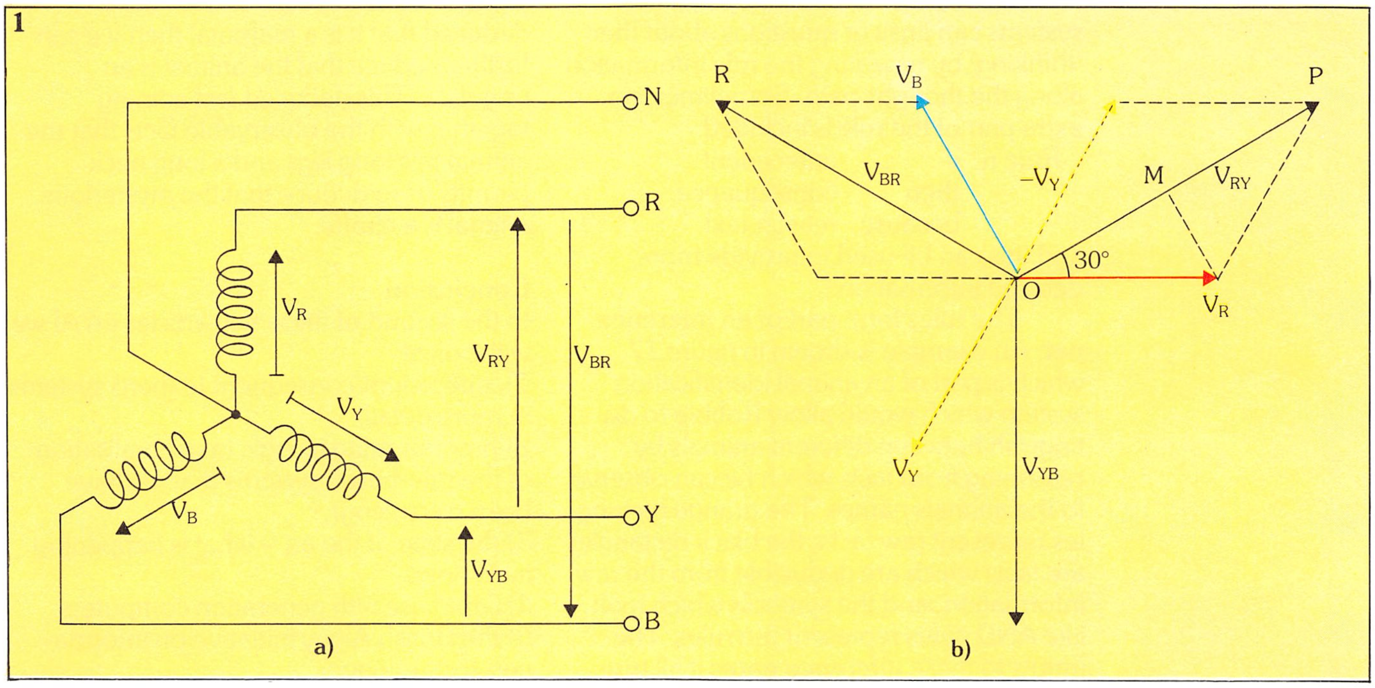
$$1.73 \times 250 = 433\text{ V}$$

Currents in a star-connected load

The currents in a star of loads connected to a

1. (a) Three-phase system connected in star formation to a three-phase supply; **(b)** phasor diagram showing voltage between each phase and neutral.

1



2

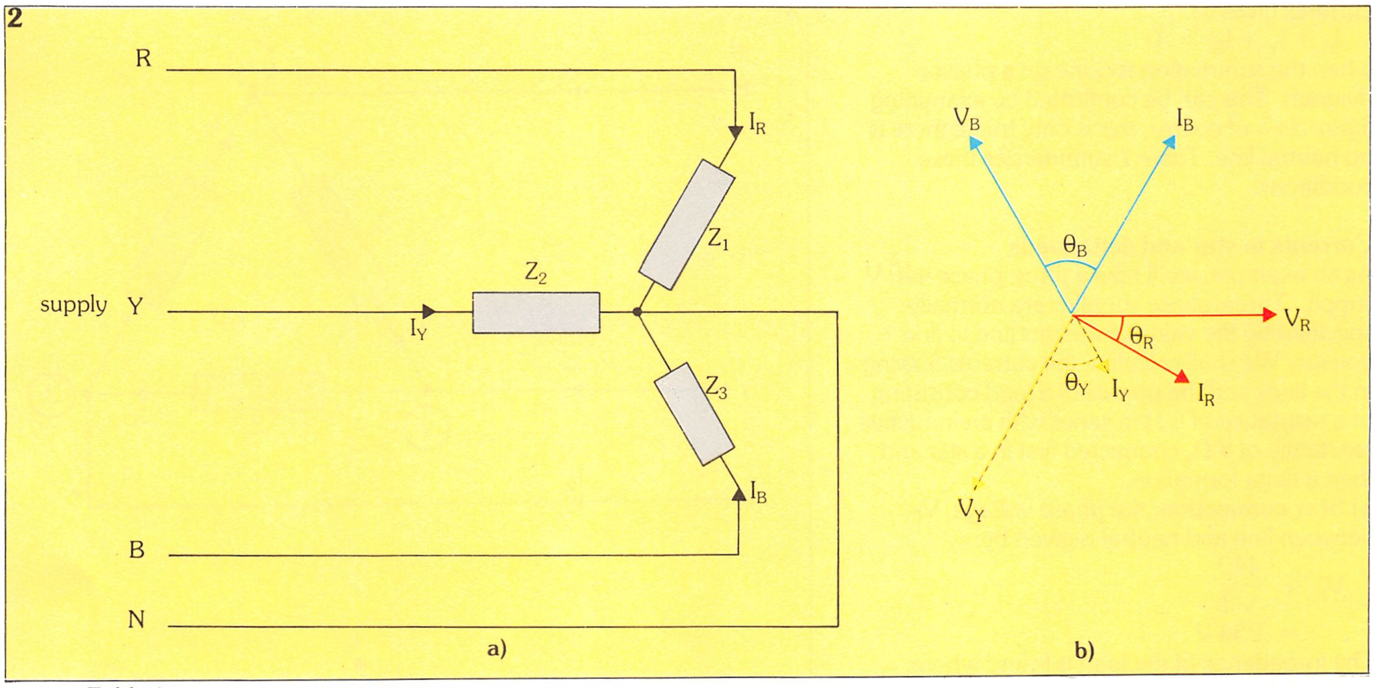


Table 1

Conditions for star or delta connection

Connection	Voltage	Current
Star	$V_L = V_P\sqrt{3}$	$I_L = I_P$
Delta	$V_L = V_P$	$I_L = I_P\sqrt{3}$

2. (a) Star of loads connected to a three-phase supply; **(b)** phasor diagram showing the phase angles between each line current and its corresponding line voltage.

three-phase supply can be found as in *figure 2a*. If the load is balanced (all the impedances are equal) the current through each impedance is the same, although the phase angles are each separated by 120°.

With an unbalanced load, the current in each impedance can be found from:

$$I_R = \frac{V_P}{Z_1}$$

$$I_Y = \frac{V_P}{Z_2}$$

$$I_B = \frac{V_P}{Z_3}$$

The phase angle between each line current and its corresponding line voltage is given by the phase angle of the appropriate impedance. These are shown in *figure 2b* for an arbitrary set of impedances.

Referring to *figure 2a*, we can also see that the currents in the three lines are equal to the three currents in the branches of the star connected load.

Currents in a delta connected load

Figure 3a shows a load of three impedances connected to a three-phase supply in a delta formation. If the three impedances are unequal, the currents I_1, I_2, I_3 may be separately calculated from the line voltages and impedances as:

$$I_1 = \frac{V_L}{Z_1}$$

$$I_2 = \frac{V_L}{Z_2}$$

$$I_3 = \frac{V_L}{Z_3}$$

These are the three-phase currents flowing in each branch of the delta connected load.

The line voltages and phase currents are shown in the phasor diagram of *figure 3b*. The currents in the three supply lines can now be calculated. At node 1 of the delta we have:

$$I_R = I_3 - I_2$$

at node 2:

$$I_Y = I_1 - I_3$$

at node 3:

$$I_B = I_2 - I_1$$

The first of these three equations can be computed by first determining $-I_2$, as shown on *figure 3c* (where we have reproduced the currents from *figure 3b*), and then adding it to I_3, completing the parallelogram and thus giving us the resultant, I_R. The other two line currents I_Y and I_B can be similarly calculated.

If the load is balanced (all the impedances are equal), we can obtain the line current in a similar way to that which was previously used to determine the phase voltages and line voltages. The line current is:

$$I_L = I_P\sqrt{3}$$

Adding together the three equations for the line

Next page

3. (a) A load of three impedances connected in delta formation to a three-phase supply; **(b)** phasor diagram showing line voltages and phase currents; **(c)** determining the line currents.

currents gives us:

$$I_R + I_Y + I_B = 0$$

when the summation is done on a phasor diagram. This can be confirmed by examining *figure 3a* – of course, this is only true if there is no neutral line. *Table 1* summarizes these conditions.

Currents in star and delta loads

As an example, we'll take a three-phase 440 V supply. Three-phase supplies are normally specified by the value of the rms line to line voltage. We shall calculate the currents flowing in the lines and the phases of a load consisting of a resistance of 6 Ω in series with an inductive reactance of 8 Ω, connected first in a star and then a delta formation.

1) Star connection: the phase voltage, V_P, between line and neutral is given by:

$$\begin{aligned} V_P &= \frac{440}{\sqrt{3}} \\ &= 254\text{ V} \end{aligned}$$

The impedance of the load is found where:

$$\begin{aligned} \text{Impedance} &= \sqrt{R^2 + X^2} \\ &= \sqrt{6^2 + 8^2} \\ &= 10\,\Omega \end{aligned}$$

The power factor:

$$\cos\phi = \frac{8}{10}$$

so:

$$\phi = 36.8°$$

The phase current in each arm of the star:

$$\begin{aligned} \frac{V_P}{Z} &= \frac{254}{10} \\ &= 25.4\text{ A} \end{aligned}$$

and this current is lagging 36.8° behind the voltage in each of the arms of the star. As *figure 1b* shows, the line voltage lags the phase voltage by 30°, so the phase current of 25.4 A thus lags the line voltage by 6.8°.

2) Delta connection: the voltage across each limb of the delta is 440 V, and as the impedance is 10 Ω, the phase current:

$$\begin{aligned} I_P &= \frac{440}{10} \\ &= 44\text{ A} \end{aligned}$$

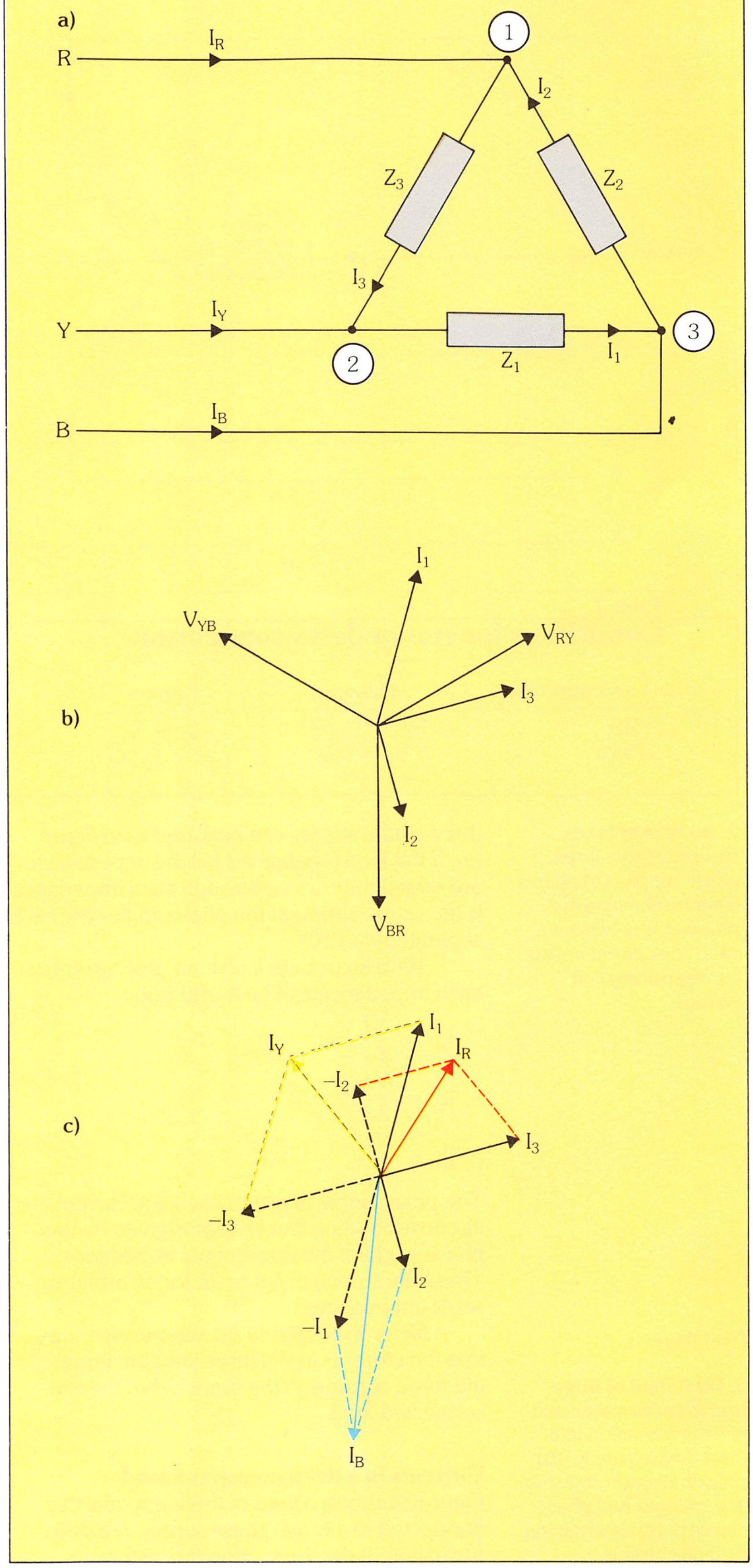
and this lags the line voltage by 36.8°.

Since this is a balanced load, the line current:

$$\begin{aligned} I_L &= I_P\sqrt{3} \\ &= 1.73 \times 44 \\ &= 76.2\text{ A} \end{aligned}$$

Again, this line current leads the phase current by 30°. We can thus see that the line current in this case is 76.2 A, lagging the line voltage by 6.8°. □

3



Reliability-2

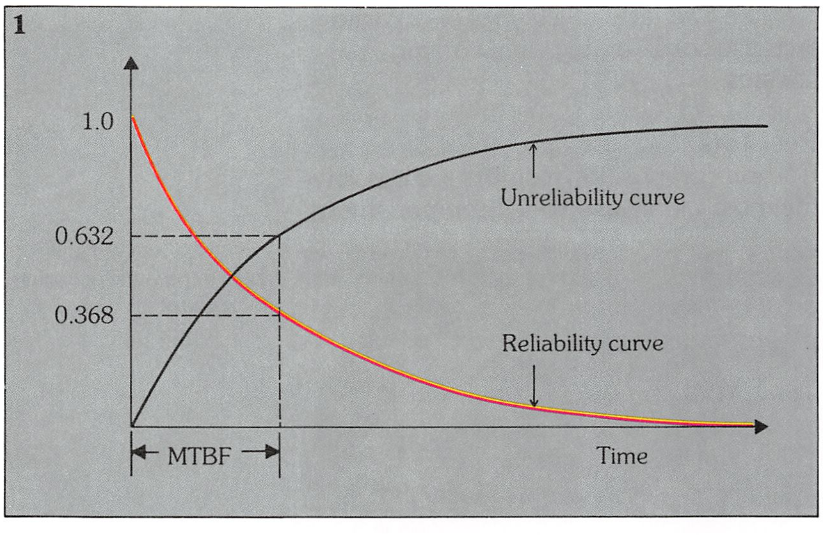
Reliability as a probability

After a manufacturer has calculated or measured (or both) a system's MTBF, the probability of a failure occurring in a period of time may then be calculated. The **probability of failure** is known as a system's **unreliability**, and may be shown on a graph, as shown in black in *figure 1*. As we can see from this graph, the probability that a failure will occur increases with time. The unreliability curve of a system, is, in fact, an exponential curve and is given by the formula:

$$P_f = 1 - \exp\left(-\frac{T}{MTBF}\right)$$

1. Reliability and unreliability curves.

1



where: P_f is the probability of failure and T is the length of time the equipment runs.

The converse of the unreliability is the **probability of survival**, or what is more commonly known as the **reliability**. The reliability curve is shown on *figure 1* in red and from it we can see that a system becomes less reliable with time, i.e. the probability of the system surviving decreases as time goes on.

Reliability may also be considered as an exponential function given by:

$$P_s = \exp\left(-\frac{T}{MTBF}\right)$$

where P_s is the reliability (i.e. probability of survival).

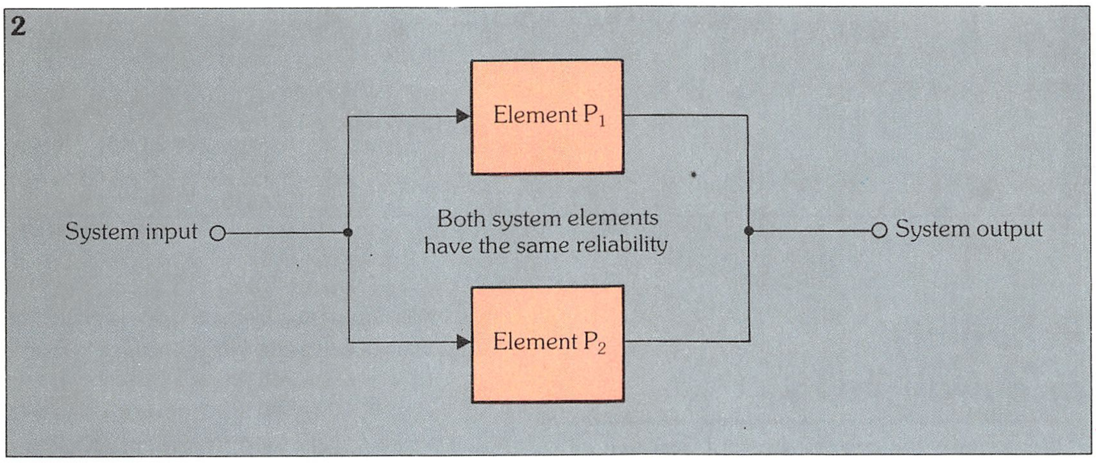
As the two exponential functions of reliability and unreliability are dependent on the MTBF of the system, we may assume that after a period of operation equal to the MTBF, the reliability curve has fallen to a value of 36.8% of its starting time, and the unreliability curve has risen to a value of 63.2% of its final value.

It is important to note that failures which are the result of poor design, development or production will not normally follow these curves. This is because the exponential reliability and unreliability curves only represent **random failures**, i.e. only those which occur in the bottom level portion of the bathtub curve. An example of a random failure is a resistor which, after many hours of operation, suddenly goes open-circuit.

Any **non-random failures**, occurring in the early period or the wear-out period of the bathtub curve, will cause the reliability and unreliability curves to alter in shape. Examples of non-random failures in the early period are caused by those components which are initially defective, and also by failures due to poor production, i.e. incorrect or faulty factory manufacture. With this knowledge, and also with a thorough understanding of the system, the manufacturer may determine the system's weak points and redesign it where necessary.

Having looked closely at the aspects which define system reliability we are now in a position to summarise the important points. These are:

1) find the predicted failure rate of the system λ_s, by calculating the failure rate of each component part and summing them;



2. Redundancy in a system improves system reliability; here the two elements are in parallel.

2) from this prediction, calculate the mean time between failures;
3) from the MTBF, calculate the system's reliability;
4) find the actual failure rate of the system by measuring a sample number of systems;
5) from the actual failure rate calculate the actual mean time between failures and hence, the actual reliability;
6) and finally, assess the differences between the predicted and actual reliability. At this stage the manufacturer must then make the decision as to which parts (if any) of the system require redesign.

Component reliability

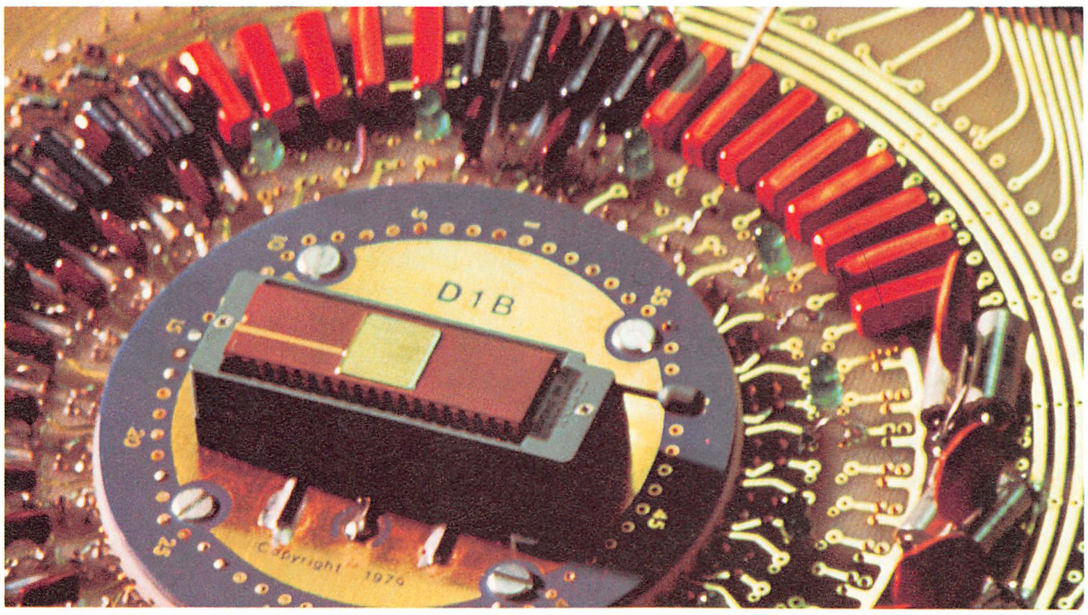
The failure mechanism of capacitors depends, to a large extent, on the type of capacitor, but generally it is due to some deterioration of the dielectric. For example, moisture in the capacitor case may cause the dielectric resistance to drop substantially, and may eventually cause a short circuit between the plates.

The capacitance of electrolytic capacitors is very temperature dependent, mainly because of the electrolyte dielectric used. At low temperatures, for example, the liquid electrolyte may freeze causing both the resistance between the plates and the capacitor's capacitance to fall. At high temperatures, the electrolyte may dry out, increasing capacitance.

Failure rates of all types of capacitors increase considerably with working stresses.

Resistors

Carbon composition resistors are possibly the most widely used of all components in



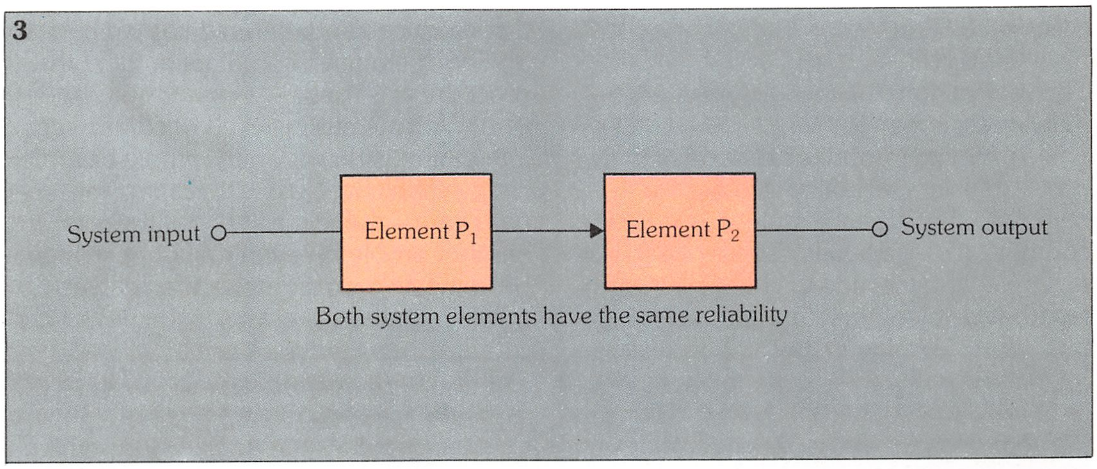
Left: apparatus for testing logic elements on a chip. (Photo: IBM).

electric systems and, fortunately, they do not exhibit high failure rates. Failures which do occur are mainly caused by increases in resistance values and open circuits.

This type of resistor has a negative temperature coefficient and so, if the temperature of the device increases, resistance decreases considerably. If overloaded, therefore, a carbon composition resistor's resistance will initially fall, but as more and more heat is dissipated ($P = V^2/R$, remember) the resistor may eventually 'burn-out' and become open-circuit.

Carbon film resistors have higher failure rates than composition resistors, due mainly to the high amount of mechanical handling required in production, however, their performance with regard to high temperatures is superior.

3. Redundant system elements in series.



Semiconductors

Failure rates of semiconductor components are generally quite low – in the region of $1 \times 10^{-7}\%$ 1000 h^{-1} – and so they have useful lives far in excess of the system. Often, the system itself will become obsolescent before the failure of any semiconductor device within it.

No particular failure mechanism is prevalent with semiconductor components and, in most cases, where a semiconductor does fail, it is usually the direct result of a failure occurring in an adjacent component. For example, the **secondary failure** of a transistor may occur due to its bias resistor short circuiting, causing excessive base current. Such a secondary failure may be caused by too high a stress on the adjacent component.

An advantage of the increasing integration of discrete components into integrated circuit form is the greater reliability of the resulting IC, compared with the discrete circuit reliability. It is this factor which has allowed complex systems such as microcomputers to be produced, with mean time between failures of many hundreds of thousands of hours.

Reliability and environment

Manufacturers of electronic systems, which are to be sold and operated throughout the world, need to take into consideration the widely varying environmental conditions in which the system may be used. These environmental conditions vary, not only in temperature and humidity, but also in biological and entomological activities, atmospheric pollution, and dust. The test chambers previously mentioned, for use when measuring a system's accelerated failure rate, may be used in some instances to measure the failure rate in harsh environments.

Improving reliability through design

Designers of electronic systems may use a number of methods of improving system reliability. One of the most important of these methods is **redundancy**, i.e. the use of duplicate components or subsystems, each of which provides the required functions.

We can see how the concept of redundancy works by considering the system of *figure 2*, in which two elements of reliability, P_1 and P_2, are connected in parallel. The relationship between system reliability and reliability of parallel elements is given by:

$$P_s = 1-(1-P_1)(1-P_2)......(1-P_n)$$

Assuming the reliability of each element is 0.9, the reliability of the system of *figure 2* is:

$$P_s = 1-(1-0.9)(1-0.9)$$
$$= 0.99$$

That is, the system reliability is greater than either of the two parallel elements.

In contrast, the reliability of a system with series elements, as in *figure 3*, is given by:

$$P_s = P_1 \times P_2 \times .....P_n$$

Thus the reliability of the system in *figure 3* is therefore:

$$P_s = 0.9 \times 0.9$$
$$= 0.81$$

That is, less than the reliability of the individual elements.

A second important method of improving system reliability is to **derate** components within the system, i.e. use the components with a lower stress rating. For example, applying lower voltage across a resistor will lower the current through it, hence reducing the power dissipated from it. Its useful life will obviously be extended and the failure rate lowered. As the resistor itself forms part of a system, and system failure rate depends on the summation of the failure rates of all components, the system failure rate will also be lowered.

Conclusion

We have seen that high reliability systems are not easily produced. A considerable amount of effort is required in design, development, production and maintenance stages to ensure the required low failure rates and long mean time between failures. Sometimes, high reliability systems are designed with low failure rates from first principles, sometimes they are first produced and then redesigned by trial and error – often a combination of both procedures is used.

With increasing levels of component integration, systems on the whole become more reliable (at least with respect to hardware – the reliability of software is another problem), even though they may be infinitely more complex than earlier, non-integrated systems.

Glossary

non-random failure	failures, occurring during the early period or the wear-out period of the bathtub curve, which are not attributable to the useful life curve
random failures	failures occurring in the bottom, level portion of the bathtub curve, i.e. due to the useful life curve
redundancy	concept of using extra components or subsystems within a system to increase the system's probability of survival
secondary failure	failure caused by the failure of a related adjacent component